RealView Debugger
User Guide

Copyright © 2002-2004 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Issue</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>April 2002</td>
<td>A</td>
<td>RealView Debugger v1.5 release</td>
</tr>
<tr>
<td>September 2002</td>
<td>B</td>
<td>RealView Debugger v1.6 release</td>
</tr>
<tr>
<td>February 2003</td>
<td>C</td>
<td>RealView Debugger v1.6.1 release</td>
</tr>
<tr>
<td>September 2003</td>
<td>D</td>
<td>RealView Debugger v1.6.1 release for RVDS v2.0</td>
</tr>
<tr>
<td>January 2004</td>
<td>E</td>
<td>RealView Debugger v1.7 release for RVDS v2.1</td>
</tr>
</tbody>
</table>

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks owned by ARM Limited. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Confidentiality Status

This document is Open Access. This document has no restriction on distribution.

Product Status

The information in this document is final, that is for a developed product.

Web Address

http://www.arm.com
Contents

RealView Debugger User Guide

Preface

About this book ..................................................................................................... viii
Feedback ........................................................................................................... xiii

Chapter 1  Starting to use RealView Debugger
1.1 Starting RealView Debugger ....................................................................... 1-2
1.2 Using RealView Connection Broker ............................................................ 1-6
1.3 RealView Debugger directories ................................................................... 1-8

Chapter 2  Working with Images
2.1 Loading images ........................................................................................... 2-2
2.2 Managing images ....................................................................................... 2-8
2.3 Working with symbols ............................................................................... 2-18
2.4 Working with multiple images ..................................................................... 2-19
2.5 Unloading and reloading images .............................................................. 2-21

Chapter 3  Controlling Execution
3.1 Submitting commands ............................................................................... 3-2
3.2 Defining execution context ....................................................................... 3-3
3.3 Using execution controls .......................................................................... 3-7
3.4 Working with the Debug menu ................................................................... 3-10
3.5 Controlling debugging .............................................................................. 3-14
Chapter 4  Working with Breakpoints
4.1  Breakpoints in RealView Debugger .............................................. 4-2
4.2  Setting breakpoints quickly .......................................................... 4-7
4.3  Using simple breakpoints .............................................................. 4-13
4.4  Setting conditional breakpoints ...................................................... 4-22
4.5  Setting hardware breakpoints ....................................................... 4-28
4.6  Using complex breakpoints .......................................................... 4-34
4.7  Using the Break/Tracepoints pane .................................................. 4-37
4.8  Disabling and clearing breakpoints ................................................ 4-43
4.9  Saving breakpoints as favorites ..................................................... 4-45

Chapter 5  Memory Mapping
5.1  About memory mapping .............................................................. 5-2
5.2  Enabling and disabling memory mapping ........................................ 5-4
5.3  Setting up a memory map .............................................................. 5-5
5.4  Viewing the memory map .............................................................. 5-7
5.5  Editing map entries ...................................................................... 5-11
5.6  Setting top of memory and stack values ......................................... 5-12
5.7  Generating linker command files for non-ARM targets .................... 5-13

Chapter 6  Working with Debug Views
6.1  Working with registers ................................................................. 6-2
6.2  Working with memory ................................................................. 6-15
6.3  Working with the stack ............................................................... 6-26
6.4  Using the call stack ................................................................. 6-32
6.5  Working with watches ............................................................... 6-36

Chapter 7  Reading and Writing Memory, Registers, and Flash
7.1  About interactive operations .......................................................... 7-2
7.2  Using the Memory/Register Operations menu .................................. 7-3
7.3  Accessing interactive operations in other ways ............................... 7-5
7.4  Working with Flash ................................................................. 7-6
7.5  Examples of interactive operations .............................................. 7-13

Chapter 8  Working with Browsers
8.1  Using browsers ................................................................. 8-2
8.2  Browsing modules and files .......................................................... 8-4
8.3  Browsing functions ................................................................. 8-7
8.4  Browsing variables ................................................................. 8-11
8.5  Specifying browser lists ............................................................ 8-14
8.6  Browsing C++ classes ............................................................... 8-16
8.7  Other routes to the browsers ...................................................... 8-18

Chapter 9  Working with Macros
9.1  About macros ................................................................. 9-2
9.2  Using macros ................................................................. 9-8
This preface introduces the RealView Debugger User Guide that shows you how to use RealView® Debugger to debug your application programs. It contains the following sections:

- *About this book* on page viii
- *Feedback* on page xiii.
About this book

This book describes how to use RealView Debugger to debug applications and images:

- a detailed description of how to use RealView Debugger to debug images using a range of debug targets, including examples
- a description of the built-in features of RealView Debugger, such as workspaces and macros
- appendixes containing reference information for the software developer
- a glossary of terms for users new to RealView Debugger.

Intended audience

This book has been written for developers who are using RealView Debugger to manage ARM® architecture-targeted development projects. It assumes that you are an experienced software developer, and that you are familiar with the ARM development tools. It does not assume that you are familiar with RealView Debugger.

This book includes an appendix that contains information for developers using RealView Debugger on Sun Solaris and Red Hat Linux.

Before you start

It is recommended that you read RealView Debugger v1.7 Essentials Guide before starting to use this book. In particular, read the chapter describing the user desktop because this contains details about menus and GUI elements used in the rest of the documentation suite.

Examples

The examples given in this book have all been tested and shown to work as described. Your hardware and software might not be the same as that used for testing these examples, so it is possible that certain addresses or values might vary slightly from those shown, and some of the examples might not apply to you. In these cases you might have to modify the instructions to suit your own circumstances.

The examples in this book use the ARM-targeted programs stored in \install_directory\RVDS\Examples\... in your RealView Debugger installation.

In general, examples use RealView ARMulator® ISS (RVISS) to simulate an ARM-based debug target. In some cases, examples are given for other debug target systems.
Using this book

This book is organized into the following chapters:

Chapter 1 Starting to use RealView Debugger
Read this chapter for details on how to start using RealView Debugger on your workstation.

Chapter 2 Working with Images
This chapter contains details on working with application programs in RealView Debugger, including how to load an image ready for debugging and how to view image details.

Chapter 3 Controlling Execution
Read this chapter for details of how to control program execution during your debugging sessions. It gives details on using the major control options and describes how to use files to keep a record of the debugging session.

Chapter 4 Working with Breakpoints
Read this chapter for details on using breakpoints to control execution of your application program. This chapter contains a full description of breakpoint options in RealView Debugger.

Chapter 5 Memory Mapping
This chapter gives details on managing memory for single processor operation during a debugging session. It describes the Process Control pane that contains a dynamic display of the current memory configuration.

Chapter 6 Working with Debug Views
Read this chapter for details of how to monitor execution of your application program by setting watches, reading registers and tracking changes to memory contents.

Chapter 7 Reading and Writing Memory, Registers, and Flash
Read this chapter for details of operations on registers contents and memory that can be accessed dynamically during a debugging session. In this way, RealView Debugger enables you to have great control over your application software.

Chapter 8 Working with Browsers
Read this chapter for details of the browsers accessible from the Code window when using RealView Debugger.
Chapter 9 Working with Macros

Read this chapter for details of how to create macros when working with RealView Debugger.

Chapter 10 Configuring Workspace Settings

RealView Debugger uses a workspace to enable you to configure your working environment and to maintain persistence information from one session to the next. You achieve this by using a workspace properties file and a global configuration file. This chapter describes the contents of these files and how to change your settings.

Appendixes and Glossary

Appendix A Workspace Settings Reference

Read this appendix for details on setting options to configure your working environment using RealView Debugger workspaces. This appendix should be read in association with Chapter 10 Configuring Workspace Settings.

Appendix B RealView Debugger on Sun Solaris and Red Hat Linux

Read this appendix for details of how to use RealView Debugger on Sun Solaris and Red Hat Linux. This appendix contains corrections and additions to the documentation suite.

Glossary

An alphabetically arranged glossary defines the special terms used in this book.

Typographical conventions

The following typographical conventions are used in this book:

*italic*  
Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

*bold*  
Highlights interface elements, such as menu names. Denotes ARM processor signal names. Also used for terms in descriptive lists, where appropriate.

*monospace*  
Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.

*monospace*  
Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.
**Further reading**

This section lists publications from both ARM Limited and third parties that provide additional information.

ARM periodically provides updates and corrections to its documentation. See http://www.arm.com for current errata, addenda, and Frequently Asked Questions.

**ARM publications**

This book is part of the RealView Debugger documentation suite. Other books in this suite include:

- *RealView Debugger v1.7 Essentials Guide* (ARM DUI 0181)
- *RealView Debugger v1.7 Project Management User Guide* (ARM DUI 0227)
- *RealView Debugger v1.7 Target Configuration Guide* (ARM DUI 0182)
- *RealView Debugger v1.7 Command Line Reference Guide* (ARM DUI 0175)

For details on using the *RealView Compilation Tools* (RVCT), see the books in the RVCT documentation suite.

For details on using RVISS, see the following documentation:


For general information on software interfaces and standards supported by ARM, see `install_directory\Documentation\Specifications\` ...

Refer to the datasheet or Technical Reference Manual for information relating to your hardware.

Refer to the following documentation for information relating to the ARM debug interfaces suitable for use with RealView Debugger:

- *RealView ICE User Guide* (ARM DUI 0155)
- *Multi-ICE® User Guide* (ARM DUI 0048)

**Other publications**

For a comprehensive introduction to ARM architecture see:

For a detailed introduction to regular expressions, as used in the RealView Debugger search and pattern matching tools, see:


For the definitive guide to the C programming language, on which the RealView Debugger macro and expression language is based, see:


For more information about the JTAG standard, see:


For more information about Oak and TeakLite processors from the DSP Group see:

Feedback

ARM Limited welcomes feedback on both RealView Debugger and its documentation.

Feedback on RealView Debugger

If you have any problems with RealView Debugger, submit a Software Problem Report:

1. Select Help → Send a Problem Report... from the RealView Debugger main menu.
2. Complete all sections of the Software Problem Report.
3. To get a rapid and useful response, give:
   - a small standalone sample of code that reproduces the problem, if applicable
   - a clear explanation of what you expected to happen, and what actually happened
   - the commands you used, including any command-line options
   - sample output illustrating the problem.
4. Email the report to your supplier.

Feedback on this book

If you have any comments on this book, send email to errata@arm.com giving:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of your comments.

General suggestions for additions and improvements are welcome.
Chapter 1
Starting to use RealView Debugger

This chapter describes how to start using RealView® Debugger ready to debug your programs. It contains the following sections:

- *Starting RealView Debugger* on page 1-2
- *Using RealView Connection Broker* on page 1-6
- *RealView Debugger directories* on page 1-8.
1.1 Starting RealView Debugger

This section describes how to start the debugger. It contains the following sections:

- Starting from Windows
- Starting from the command line
- Setting environment variables on page 1-5.

1.1.1 Starting from Windows

To start RealView Debugger:

1. Select Programs → ARM from the Windows Start menu.
2. Select RealView Developer Suite v2.1 → RealView Debugger v1.7 from the menu.

1.1.2 Starting from the command line

The syntax for the command-line method of starting RealView Debugger is as follows:

```
rvdebug.exe [-bat|-cmd][-install=pathname][-user=name][-Home=pathname]
[-aws=pathname][-aws=-][-exec image_pathname]
[-inc pathname][-jou pathname][-log pathname][-s pathname][-nologo]
```

where:

- `bat` Runs a RealView Debugger session in batch mode, that is without any user interaction.
  
  Use this with `-inc` to run a script file containing commands.
  
  Can be replaced with `-b`.

  _______ Note _______
  
  Do not use `-b` without `-inc`. If you use only `-inc`, the script file is run with the GUI enabled.

- `cmd` Runs the command-line debugger only to use CLI commands to carry out debugging tasks. This enables you to interact with the debugger without using the RealView Debugger GUI.

- `install` Specifies the installation directory where this differs from the default installation. This is then used to define the location of the default RealView Debugger home directory when the debugger runs for the first time.

  This must be used if the environment variable RVDEBUG_INSTALL is not set.
-user Specifies the user ID in the RealView Debugger home directory used for the debugging session. Where this is not specified, the default Windows login is used.

See Defining the home directory on page 1-8 for details.

-Home Specifies a RealView Debugger home directory used for the debugging session. If the specified directory does not exist, a new one is created. Where this is not specified, the default directory is used.

--- Note ---

This option is only available under Windows. On Sun Solaris and Red Hat Linux platforms, RealView Debugger always uses the home directory as specified by the $HOME environment variable.

--- See Defining the home directory on page 1-8 for details.---

-aws Runs a RealView Debugger session with the specified workspace. This overrides any workspace specification that was stored when the previous session ended.

Use -aws=-- to start without a workspace.

-exec Specifies the image loaded when RealView Debugger runs. The image specification can also include target details and image arguments.

-inc Runs a RealView Debugger session with the specified include file.

Use -inc:

• in batch mode in association with the -bat setting, to execute the commands contained in the file and then exit the debugger
• in command-line mode in association with the -cmd setting, to execute the commands contained in the file and then leave the debugger running ready to continue the debugging session
• in GUI mode on its own, to execute the commands contained in the file during a debugging session.

-jou Runs a RealView Debugger session with the specified journal file open for writing. Can be replaced with -j.

-log Runs a RealView Debugger session with the specified log file open for writing. Can be replaced with -l.

-s Runs a RealView Debugger session with the specified STDIOlog file open for writing.
Starting to use RealView Debugger

-no logo

Runs a RealView Debugger session without displaying a Windows splash screen.

Examples

To start RealView Debugger and specify an installation directory, where RVDEBUG_INSTALL is not set:

`program_directory\bin\rvdebug.exe -install="E:\ARM\RVD"`

If RVDEBUG_INSTALL is set then the -install overrides it.

To start RealView Debugger and specify a home directory, where RVDEBUG_HOME is not set:

`program_directory\bin\rvdebug.exe -Home="D:\ARM\RVD\home\my_user_home"`

To start RealView Debugger and specify a workspace:

`program_directory\bin\rvdebug.exe" -aws="D:\ARM\RVD\home\my_user_name\friday_test.aws"

To start RealView Debugger without loading a workspace:

`program_directory\bin\rvdebug.exe -aws=-`

To start RealView Debugger with a log file open for writing:

`program_directory\bin\rvdebug.exe -log
"D:\ARM\RVD\home\my_user_name\test_files\my_log.log"

To start RealView Debugger with a specified image loaded that takes two arguments:

`program_directory\bin\rvdebug.exe -exec "C:\RVD\images\my_image.axf;;arg1 arg2"

In these examples, your program_directory might be a default location such as install_directory\RVD\Core\1.7\build\platform.

Getting more information

To find more information on operations available from the command line, see:

- Chapter 2 Working with Images for details on loading images.
- Chapter 3 Controlling Execution for details on using log and journal files.
- Chapter 10 Configuring Workspace Settings for details on workspaces.
1.1.3 Setting environment variables

User-defined environment variables can be set to configure RealView Debugger. Set RVDEBUG_INSTALL or RVDEBUG_HOME to override the default locations, for example to specify:

- an installation directory that differs from the default, set:
  RVDEBUG_INSTALL=D:\Program Files\ARM\RVD

- a home directory that differs from the default, set:
  RVDEBUG_HOME=E:\Program Files\ARM\RVD\my_home

To specify a shared location for RealView Debugger target configuration files, set:

RVDEBUG_SHARE=H:\ournet\devel\rvd\shared
1.2  Using RealView Connection Broker

Target vehicles can reside on the same workstation as RealView Debugger or any other workstation on your network. These services are handled by the RealView Connection Broker, rvbroker.exe.

RealView Connection Broker operates in two modes:

- **Local** Operating as RealView Simulator Broker, this runs on your local workstation and enables you to access targets on the local workstation.

- **Remote** Operating as RealView Network Broker, this runs on a remote workstation and makes specified targets on that workstation available to other workstations connected to the same network.

Local host simulators are available immediately from the Connection Control window. If you expand the Simulator Broker vehicle, ready to connect to a simulator, RealView Debugger starts RealView Connection Broker in local mode to manage your connection.

See the chapter describing configuring custom connections in *RealView Debugger v1.7 Target Configuration Guide* for examples of how to set up your own connections.

1.2.1  Starting RealView Network Broker

Any remote workstation that is to give access to simulators or emulators must be running RealView Connection Broker in remote mode, that is RealView Network Broker. This can be started in two ways:

- if the remote workstation is running Sun Solaris or Red Hat Linux and the rsh command is available at the local workstation, the local workstation can start RealView Network Broker on the remote workstation
- if the remote workstation is running Windows, RealView Network Broker must be started explicitly on that workstation.

If you are using a remote Windows workstation to access simulators or emulators, start RealView Network Broker:

1. Log onto the remote workstation.
2. Select Programs → ARM from the Windows Start menu.
3. Select RealView Developer Suite v2.1 → RealView Network Broker v1.7.

The syntax for the command-line method of starting RealView Connection Broker in local or remote mode is as follows:
rvbroker.exe -install=pathname 0 remote

where:

-install Specifies the installation directory where this differs from the default installation.
This must be used if the environment variable RVDEBUG_INSTALL is not set.

0 Specifies that the default TCP/IP port is used.

remote Specifies that the TCP/IP port is used to make this workstation visible to other network users.

For example, to start RealView Network Broker on a Windows workstation and specify an installation directory, where RVDEBUG_INSTALL is not set:

rvbroker.exe -install="E:\ARM\RealView Debugger" 0 remote

--- Note ---
If you end a debugging session, and close down RealView Debugger, this does not terminate RealView Network Broker on the remote workstation. This must be shut down explicitly.

---
To access a remote host simulator or emulator using RealView Network Broker you must define the location of the remote workstation in your target configuration settings. The chapter describing working with remote targets in RealView Debugger v1.7 Target Configuration Guide includes examples of how to set up your own connections.
1.3 **RealView Debugger directories**

RealView Debugger must be able to identify the installation directory and a home directory so that it can locate files and store updated files or user configuration details. This section describes:

- **Defining the installation directory**
- **Defining the home directory**
- **Using the examples directories** on page 1-9.

1.3.1 **Defining the installation directory**

RealView Debugger must be able to identify the installation directory so that it can locate user files and configuration files. It uses the following to define the installation directory (in order of priority):

1. The `-install` command line argument, where used.
2. The `RVDEBUG_INSTALL` environment variable, where set.
3. The default location as defined by the root installation.

1.3.2 **Defining the home directory**

RealView Debugger requires a home directory to store user-specific settings and configuration files. This is not the same as your Windows home directory.

The location of this directory depends on the environment variables set, and the command line arguments used, when RealView Debugger starts. It uses the following tests to define the home directory:

1. The `-Home` command line argument (only available under Windows), if used.
2. The `RVDEBUG_HOME` environment variable, if set.
3. The `-user` command line argument, if used. This is then used to specify the user ID in the home directory, for example set `USER=my_user_name` to specify the home directory `...\home\my_user_name`.
4. Your default Windows login, for example `...\home\WinLogID`.

If your Windows login ID contains spaces, these are converted to underscores. Any ID longer than 14 characters is automatically truncated.
Starting to use RealView Debugger

Because you can choose the home directory, the installation directory and your user name, the RealView Debugger home directory is defined in this book as being in a default location install_directory\RVD\Core\...\home\user_name, where user_name is your Windows login ID. This means that your files might be stored in places other than those given in the examples.

For details on the files that are stored in the RealView Debugger home directory see the online help topic Where is information stored?

1.3.3 Using the examples directories

Various demonstration projects are supplied as part of the RealView Debugger root installation. These contain programs in the form of ARM® assembly language, C, or C++ source code files. These projects are stored in install_directory\RVDS\Examples\....

The root installation also includes demonstration projects, and associated files, for working with Flash. These are in install_directory\RVD\Core\...\Flash\examples\....
Starting to use RealView Debugger
Chapter 2
Working with Images

This chapter describes how to manage images during a RealView® Debugger debugging session. It contains the following sections:

- *Loading images* on page 2-2
- *Managing images* on page 2-8
- *Working with symbols* on page 2-18
- *Working with multiple images* on page 2-19
- *Unloading and reloading images* on page 2-21.
2.1 Loading images

If you have started RealView Debugger, as described in Chapter 1 Starting to use RealView Debugger, you can begin to use many features of the debugger, for example editing source code and building projects. However, to begin debugging images you must connect to a suitably configured debug target.

RealView Debugger uses a board file to access information about the debugging environment and the debug targets available to you, for example how memory is mapped. See RealView Debugger v1.7 Target Configuration Guide for details of how to customize your targets using your board file.

You can start to use RealView Debugger with the default board file installed as part of the root installation without making any further changes.

Select File → Connection → Connect to Target... from the main menu to display the Connection Control window to make your first connection. For details on using this window, see the chapter describing getting started in RealView Debugger v1.7 Essentials Guide.

If you have started RealView Debugger and connected to a debug target, you can load an image to begin your debugging session. This section describes different ways to load an image to your debug target and how to monitor the loading operation:

- Loading from a user-defined project on page 2-3
- Using the Load File to Target dialog box on page 2-3
- Loading from the Process Control pane on page 2-5
- Quick loading on page 2-5
- Loading from the command line on page 2-6
- Loading and runtime visualization on page 2-7.

The examples in this section assume that you are using a Typical installation and that the software has been installed in the default location. If you have changed these defaults, or set the environment variable RVDEBUG_INSTALL, your installation differs from that described here.

Note

When you are using RealView ARMulator® ISS software simulator to simulate an ARM® architecture-based debug target:

- You must load an image, or write to the PC, to begin execution.
- Loading an image does not automatically send a reset. To reset at the same time as an image load, send a RESET command before you load, or reload, the image.
2.1.1 Loading from a user-defined project

Where you have created a user-defined project, it is recommended that you open the project first to load and debug the associated image, or images. Opening the project enables you to access the project properties, save new settings, or make changes to the build model.

With a user-defined project open, for example \dhrystone\dhrystone.prj, from the install_directory\RVDS\Examples\... directory, click on the hyperlink in the File Editor pane to load the associated image.

--- Note ---
If you load an image, built as part of a user-defined project, without opening the project, this does not give you access to all the project properties because these are unknown to RealView Debugger. In this case, RealView Debugger creates an in-memory project, or uses the saved auto-project file (see Working with auto-projects on page 2-12 for details).

2.1.2 Using the Load File to Target dialog box

Select File → Load Image... from the Code window main menu to load an image to a processor for execution. This displays the Load File to Target dialog box shown in Figure 2-1.

![Load File to Target dialog box](image)

Figure 2-1 Load File to Target dialog box
This dialog box contains controls to configure the way the image is loaded for execution:

**Symbols Only**

By default, any object file loaded from this dialog box also loads the symbols. If you want to load only the symbols then select this check box, for example when you are working with ROM images.

If the program was initially compiled without a symbol table then you must recompile the program before loading only the symbols.

See *Working with symbols* on page 2-18 for more details.

**Replace Existing File(s)**

By default, loading a new image overwrites any image currently loaded to the target.

If you are working with multiple applications, use this check box to carry out separate loads of associated modules such as an RTOS and associated applications.

**Target Name:**

Use this field to enter the target name, where supported.

A name entered here is then used as the argument to a LOAD command (see *Specifying the load instruction* on page 2-7).

**Arguments:**

Use this field to enter a space-separated list of arguments to the image.

Entries in this field create an arguments list used with the LOAD command (see *Specifying the load instruction* on page 2-7).

**PC**

When you load an image to the debug target you can optionally set the **Program Counter** (PC):

**Auto-Set PC**

Selected by default, this control defines the location of the PC when you load an image. RealView Debugger tracks the state of the other check boxes on this dialog box and sets the PC at the normal entry point, if you select the check box **Replace Existing File(s)**.

Unselect the **Auto-Set PC** check box to have control over the PC when you load an image.

**Set PC to Entry point**

Where selected, RealView Debugger sets the PC at the start address specified in the object module.
This is the default if you select both:

- **Auto-Set PC**
- **Replace Existing File(s).**

Unselect the **Set PC to Entry point** check box to prevent the load command setting the PC.

--- **Note** ---

Controls used here, for example setting the PC, might be overridden by load settings elsewhere, for example as specified in your project or target configuration settings.

### 2.1.3 Loading from the Process Control pane

If you have started RealView Debugger and are connected to a debug target, you can load an image for execution from the Process Control pane:

1. Select **View → Pane Views → Process Control Pane** from the default Code window main menu to display the Process Control pane.
   
   Whilst there is no image loaded, the pane only shows details about the debug target processor and the current location of the PC.

2. Right-click on the top line, the **Process** entry, to display the **Process** context menu.

   Whilst there is no image loaded, you can also display this menu from the **Image** entry.

3. Select **Load Image...** to display the Load File to Target dialog box.

4. Complete the entries in the dialog box, described in **Using the Load File to Target dialog box** on page 2-3, to load the required image.

### 2.1.4 Quick loading

You can load an image by dragging the appropriate executable file, with the `.axf` extension, and dropping it into the File Editor pane. If successful, this is the same as loading the image using the Load File to Target dialog box with the default settings (shown in Figure 2-1 on page 2-3), that is the load auto-sets the PC and overwrites any existing image on the debug target.

--- **Note** ---

Ensure that the current connection, shown in the Code window title bar, matches the processor type of the image you are trying to load. If they do not match the load fails.
2.1.5  Loading from the command line

You can start RealView Debugger from the command line and specify an image to load automatically. The syntax for loading an image this way is as follows:

```
rvdebug.exe -exec pathname
```

where `pathname` specifies the image loaded and can also include target details and image arguments (see Specifying the load instruction on page 2-7 for details).

If the pathname includes spaces, it must be enclosed in quotes, for example:

```
"C:\Program Files\ARM\RVD\bin\rvdebug.exe" -install=C:\rvd
-exec "C:\rvd\my images\my_image.axf"
```

This starts RealView Debugger, specifies an installation directory, and issues a `load/pd/r` command to load the named image to your debug target. Any error messages appear in a dialog box, specified by `/pd`. This command replaces any image currently loaded on the chosen target, specified by `/r`.

Note
For details on running RealView Debugger from the command line see Chapter 1 Starting to use RealView Debugger.

Making a connection

If you are connected to your debug target, starting RealView Debugger in this way loads the specified image to the target and updates the Code window. This is the same as Using the Load File to Target dialog box on page 2-3.

If you are not connected to your debug target before starting RealView Debugger, loading an image from the command line starts the debugger and then displays a prompt box, shown in Figure 2-2, for you to complete the connection.

![Figure 2-2 Connection prompt](image)
Click either:

**Yes**  
Causes the debugger to wait until you connect to your debug target. The image is then loaded to the connected target.

**No**  
Starts the debugger but cancels the image loading operation.

**Specifying the load instruction**

If you are loading an image from the command line, you can pass arguments to the image and specify the target name that is passed to the image loader. The syntax is as follows:

```
rvdebug.exe -exec image.axf;target_name;[arg1 arg2 ...]
```

where:

- **image.axf**  
  Specifies the image loaded.

- **target_name**  
  Specifies the target name, where supported.

- **arg1 arg2 ...**  
  Specifies an optional, space-separated, list of arguments to the image.

Specifying the target name depends on the underlying OS support and your debug target. For example, if you are using an RTOS image loader, then this target name is passed to the loader. In the example below, you are using the debugger built-in loader and so specifying target name has no effect and can be omitted:

```
C:\rvd\bin\rvdebug.exe -exec "C:\rvd\images\my_image.axf;;arg1 arg2 arg3"
```

**Note**  
Spaces must not be included between the argument and the qualifier. Where an arguments list is given, quotes must be used.

For details on debugging applications using an RTOS see the chapter describing RTOS support in *RealView Debugger v1.7 Extensions User Guide*.

### 2.1.6 Loading and runtime visualization

As you load an image to your debug target, the Code window Status line shows the progress of the load and gives an indication of the percentage complete.

The State group, on the Actions toolbar, shows the runtime state of the debug target. Where an image is loaded but not executing, this shows **Stopped**. A moving progress indicator signals an application is running.
2.2 Managing images

This section describes how to manage your application files in the Code window. It contains the following sections:

- Viewing image details in the Code window
- Viewing image details in the Process Control pane on page 2-10
- Working with auto-projects on page 2-12
- Working with user-defined projects on page 2-16.

The examples in this section assume that:

- you are using a Typical installation and that the software has been installed in the default location. If you have changed these defaults, or set the environment variable RVDEBUG_INSTALL, your installation differs from that described here.

- you are licensed to use multiprocessor debugging mode. However, where given, multiprocessor examples are only used for illustration. If you are licensed to work in single-processor mode your Code window differs from that shown here, for example some toolbar buttons are not available to you, but all the examples work as described.

2.2.1 Viewing image details in the Code window

If an image is successfully loaded to the target processor, the Code window is updated, shown in Figure 2-3 on page 2-9.
Figure 2-3 Code window with image loaded

--- Note ---

In this Code window Text Coloring is enabled by default and line numbering is turned on by selecting Edit → Editing Controls → Show Line Numbers.

RealView Debugger updates the default views in the side pane and middle panes row and, where known, displays information about the new image. Because you have not started debugging, other panes are empty.

When you load an image with symbols, as here, RealView Debugger searches for corresponding source files and displays these as tabs in the File Editor pane. The Src tab acts like a button to display the current source if it is available. In this example, click on the Src tab to display the source-level code view.

The image was loaded with the Auto-set PC option set and so execution control is located at the default entry point. This is indicated by a box at line 78, colored red by default.

Click on the Dsm tab to show the disassembly-level view.

See Code views on page 3-3 for more details on using these tabs.
2.2.2 Viewing image details in the Process Control pane

Select View → Pane Views → Process Control Pane to display the Process Control pane, shown in Figure 2-4.

![Figure 2-4 Image details in the Process Control pane](image)

The Process Control pane contains tabs:

**Process**

- Displays details of the target processor or, in multiprocessor debugging mode, the current process.

  See Working with processes for details.

**Map**

- Displays the memory mapping for the target processor, or the current process, to enable you to change the map settings.

  See Chapter 5 Memory Mapping for details on using this tab.

The tabs displayed in the Process Control pane depend on the debugging mode that you are licensed to use and your current debugging environment. For example, when debugging multithreaded applications, a Thread tab is displayed. See the chapters describing RTOS support and multiprocessing in RealView Debugger v1.7 Extensions User Guide for more details.

Working with processes

The Process Control pane shows details about each connection known to RealView Debugger. If you are debugging a single process application, use the Process tab to see the processor details, project details, and information about any image(s) loaded onto the debug target, for example:

- image name
- image resources, including DLLs
- how the image was loaded
- load parameters
associated files.

Use context menus in the Process tab to:

- reset your target processor (where supported)
- load, unload, and reload images, and refresh symbols
- manage settings for auto-projects and user-defined projects
- scope to a specified source file.

In the example in Figure 2-4 on page 2-10, you can see the entries:

**Current process**
- Shows the target processor and the current state of any running process.
- Where the process is stopped, as here, this shows the location of the PC.
- Where the process is executing, this changes to Run.

**Image**
- Details the loaded images:
  - **Load** For each image, a check box indicates the load state and what has been loaded, that is image, symbols, or both.
  - **Project** Shows that the project associated with the connection is either a real, user-defined project file (shown by the project name) or an auto-project (shown by <Auto>).
  - **Settings** Shows where project settings are stored. These might be from a disk file (shown by <Saved>) or from an in-memory auto-project (shown by <Not Saved>).
  - **Sources** These are either the sources making up the project, sources extracted from the makefile used in the build, or sources from the loaded image.
    - Depending on the type of project, right-click on this entry to display a context menu to specify how sources are collected.

**Working with source files**

When working with entries in the Process Control pane, you can use type ahead facilities to locate files. This is especially useful where your project specifies a large number of source files. For example, type the first letter of the source file that you want to view. RealView Debugger expands the Sources entry and locates the first matching occurrence. When using this feature, the type ahead buffer is case insensitive and is limited to 128 characters. Do one of the following to clear the buffer:

- select a different item
Getting more information about an entry in the Process tab

Right-click on an entry in the Process tab to see the context menu associated with that entry. Select Properties to see a text description of the item under the cursor.

___ Note ___
The options available from the context menu depend on which entry is selected and the current state of the process or processor.

2.2.3 Working with auto-projects

An auto-project is a custom, image control, project that holds project settings where the build model is unknown.

When you load an image directly to a debug target, RealView Debugger checks to see if an auto-project file exists for the image in the same location. Where an auto-project exists, RealView Debugger opens it and then uses it to load the specified image. Where no auto-project exists, RealView Debugger creates an in-memory auto-project to use in this session.

If, for example, you load the image dhrystone.axf, in install_directory\RVDS\Examples\..., RealView Debugger looks for the corresponding auto-project file dhrystone.axf.apr, in the same location. Where no auto-project exists, RealView Debugger creates an in-memory auto-project, named dhrystone. The Process tab is then updated with the project details, shown in Figure 2-5 on page 2-13.
RealView Debugger gives you the option to save the in-memory settings to a file to use next time the image is loaded or as the basis of a new user-defined project.

Viewing project settings

You can view settings for the in-memory auto-project just like a user-defined project:

1. Right-click on the Project entry, to display the Project context menu. You can also display this menu from the Settings entry.

2. Select Project Properties... to display the Project Properties window where you can view the project settings. These are derived from the image details or created using defaults by RealView Debugger.

3. Select File → Close Window to close the Project Properties window without making any changes.

Note

Do not leave the Project Properties window open when you are working with the debugger. When searching for source files, RealView Debugger updates the project properties as necessary. This fails if the window is already open (see Searching for source files on page 3-17 for details).
Changing project settings

You can change load settings for an image where you do not have a user-defined project by defining actions in the auto-project and then the saving the file for use next time the image loads. You can specify commands to execute when the project opens and/or closes, or runtime controls that define the image environment.

Note
Changing auto-project settings might not take effect until the next time the image is loaded and executed. Reload an image to implement any new settings.

You can change settings for the in-memory auto-project just like a user-defined project:

1. Right-click on the Project entry, to display the Project context menu.
2. Select Project Properties... to display the Project Properties window.
3. Expand the PROJECT group to see the project settings, shown in Figure 2-6.
4. Expand the SETTINGS group to see the image settings.

Figure 2-6 Changing auto-project settings

Here you can see the Command_Open_Close group and other project settings.

4. Expand the SETTINGS group to see the image settings.

   Figure 2-6 shows the Image_load group and other image settings, such as breakpoints and runtime controls.

5. Right-click on the Image_load group and select Explore to see the group contents in the right pane.

6. Right-click on the Set_pc entry and select never from the options.
7. Select File → Save and Close to save your changes and close the Project Properties window.

To return the setting to the default:

1. Display the Project Properties window.
2. Right-click on the entry to display the context menu.
3. Select Reset to Default to restore the setting.
4. Select File → Save and Close to save your changes and close the Project Properties window.

Note

There are full descriptions of the entries in the Project Properties window, with details of the available options, in the RealView Debugger online help topic Changing Project Properties.

Saving project settings

Save the auto-project so that the new settings are used when you next load the image. There are two ways to save an auto-project:

- In the Project Properties window, select File → Save Changes to close the window and save any changes to the file dhrystone.axf.apr.
- In the Process Control pane, Process tab, right-click on the Project <Auto> entry and select Save from the Project context menu to save the file dhrystone.axf.apr.

You can delete a saved auto-project so that the file is removed from your disk:

1. Right-click on the Project entry, to display the Project context menu.
2. Select Delete Auto-Project File to remove the saved file.

Closing auto-projects

To close an auto-project, right-click on the Project <Auto> entry in the Process tab and select Close from the Project context menu. If you close the auto-project associated with a loaded image, this immediately unloads the image and removes all image details from the debugger. If you close an auto-project, RealView Debugger executes any close commands associated with the project.
2.2.4 Working with user-defined projects

With a user-defined project open, right-click on the Project entry to display the Project context menu, shown in Figure 2-7.

![Figure 2-7 Project menu](image)

This menu enables you to view details of your project, make changes to project settings, and to perform selected components of the build model following changes to project files.

See the description of the Project menu in the chapter describing working with projects in RealView Debugger v1.7 Project Management User Guide for full details on these options.

Closing user-defined projects

To close a user-defined project where the associated image is loaded, right-click on the Project entry in the Process tab and select Close from the Project context menu.

RealView Debugger gives you the option to unload the image when the project closes.

If you choose to unload the image, RealView Debugger completes the operation, closes the project, and then executes any close operations associated with the project.

If you do not unload the image, the debugger:

1. Closes the user-defined project.
2. Executes any close commands associated with the project.

Note

You can also use the Project menu from the Code window main menu to close auto-projects.

See the chapter describing working with multiple projects in RealView Debugger v1.7 Project Management User Guide for more details on working with auto-projects.
3. Either:
   - opens the saved auto-project file, where one exists for the image
   - creates an in-memory project where no saved auto-project exists.

It is not necessary to reload the image following these actions.

——— Note ————

You can also use the Project menu from the Code window main menu to close user-defined projects in the same way.

———

See the chapter describing working with projects in RealView Debugger v1.7 Project Management User Guide for more details on closing your projects.
2.3 Working with symbols

An executable image contains symbolic references, such as function and variable names, as well as the program code and data.

If you select the **Symbols Only** check box, on the Load File to Target dialog box (see Figure 2-1 on page 2-3), the symbolic references are loaded into the debugger without loading any code or data to the target. You might want to do this if the code and data are already present on the debug target, for example in a ROM device or where you are working with an RTOS-enabled target.

You can choose to refresh the symbol data for a loaded image during your debugging session. There are two ways to do this for the current process, depending on the number of images loaded:

- select **File → Refresh Symbols** from the Code window main menu
- use the Process Control pane:
  - right-click on the Image entry to display the **Image** context menu
  - select **Refresh Symbols** from the available options.

Note

When an image is loaded with symbols, the symbol table is recreated. This automatically deletes any macros because these are stored in the symbol table.
2.4 Working with multiple images

RealView Debugger provides the option to load multiple images to the same debug target, that is where there is only a single connection. This enables you to load, for example, both an executable image and an RTOS at the same time.

To load two images to the same debug target:

1. Load the first image, for example hello.axf, in the usual way.
2. Load a second image to the same target, for example demo.axf. The two images must not overlap in memory.
   
   **Note**
   
   Remember to unselect the **Replace Existing File(s)** check box.

3. Select **View → Pane Views → Process Control Pane** from the default Code window main menu to display the Process Control pane.
4. Expand the display to see the process details, shown in Figure 2-8.

   ![Figure 2-8 Multiple images in the Process Control pane](image)

In this example, RealView Debugger:

- creates an in-memory auto-project for hello.axf
- binds hello.axf.apr to the current connection (using default binding)
- creates an in-memory auto-project for demo.axf
- does not bind demo.axf.apr, because there is no connection available
- updates the Code window title bar to show the active project (hello)
- updates the title bar of the floating Process Control pane to match.

For information on projects and project binding see *RealView Debugger v1.7 Project Management User Guide*. 
Because neither image is currently executing, the Process entry shows the current location of the PC, auto-set when the first image was loaded. You can move the PC manually to start debugging or reload the image that you want to test. For information on changing the PC see:

- Chapter 3 Controlling Execution for details on setting scope.
- Chapter 6 Working with Debug Views for details on changing register entries.

**Note**

If you are working with multiple images, you must set a manual breakpoint at the entry point for any images loaded above 0x8000. This ensures that RealView Debugger is able to debug these images in the usual way.

See the chapter describing multiprocessing in *RealView Debugger v1.7 Extensions User Guide* for more details on working with the Process Control pane in multiprocessor debugging mode.
2.5 **Unloading and reloading images**

This section describes how to unload and reload images without ending your debugging session. It contains the following sections:
- Resetting your target processor
- Unloading an image
- Reloading an image on page 2-22.

2.5.1 **Resetting your target processor**

Where supported by your debug target, you might want to reset your target processor during a debugging session. The reset might be hard or soft depending on the processor type. See your processor hardware documentation for details.

If the processor chosen for reset has an image loaded then this might be unloaded on reset. The image can be reloaded as described in *Reloading an image* on page 2-22.

To reset a processor:

1. Select View → Pane Views → Process Control Pane from the default Code window main menu to display the Process Control pane.
2. Right-click on the Process entry to display the Process context menu.
3. Select Reset Target Processor to perform the reset.

2.5.2 **Unloading an image**

You do not have to unload an image from a debug target before loading a new image for execution. To load over an existing image, ensure that the Replace Existing File(s) check box is checked on the Load File to Target dialog box, shown in Figure 2-1 on page 2-3. This automatically removes all details about the first image from RealView Debugger.

Use the Process Control pane if you do want to unload an image explicitly:

1. Right-click on the Image entry to display the Image context menu.
2. Select Unload from the available options.

This is the same as clicking on the Load check box to unload the image.

If there are any source file tabs currently displayed in the File Editor pane, the Src tab is brought to the top automatically when you unload because there is no known context. The open files remain available for further editing. See Code views on page 3-3 for more details on using these tabs.
Unloading an image does not affect target memory. It unloads the symbol table (and any macros) and removes debug information from RealView Debugger. However, the image name is retained for reloading and the associated auto-project, or user-defined project used to load the image, is maintained.

Note

Unloading an image does not close the in-memory auto-project, associated with the image, or save any changes to the auto-project. This enables you to modify these settings, and save, ready for the next time you load (or reload) this image. However, if you close the auto-project explicitly, RealView Debugger performs an image unload.

Deleting image details

To remove all details about an image after you have unloaded it, right-click on the Image entry in the Process Control pane and select Delete Entry from the context menu. If there is an auto-project associated with the image, this closes. If you opened a user-defined project to load the image, this does not close.

Disconnecting with an image loaded

If you disconnect with an image loaded, this removes debug information from RealView Debugger and so clears pane contents from your Code window. This does not close the user-defined project used to load the image, or any auto-project associated with the image. You can reload the image if you reconnect.

However, if you close the project explicitly, you will have to load the image again after you reconnect because all image details have been removed.

2.5.3 Reloading an image

During your debugging session you might have to edit your source code and then recompile. Following these changes, you must either:

- load the previously unloaded image
- reload the image.

Reloading refreshes any window displays and updates debugger resources.

To reload an image:

- Select File → Reload Image to Target from the Code window to reload an updated image to your debug target.

An image that has been unloaded cannot be reloaded using this option. Instead you must load the image again using File → Load Image....
• Click the **Reload Image** button on the Actions toolbar.

• Select the **Load** check box in the Process Control pane.
  This is the same as double-clicking on the Load entry.

RealView Debugger uses auto-project settings to load an image and these are automatically used when you reload the image, or when you select it from the Recent Images list.

If you use the Load File to Target dialog box to enter a space-separated list of arguments to the image, these are not used when you reload (see *Using the Load File to Target dialog box* on page 2-3 for details). To reuse the arguments either:

• Select **File → Recent Images** to reload the image from the Recent Images list.

• Click the **Set PC to Entry** button on the Actions toolbar to submit a **RESTART** command.
Chapter 3
Controlling Execution

There are several ways to control program execution from the RealView® Debugger Code window. These are described in the following sections:

- Submitting commands on page 3-2
- Defining execution context on page 3-3
- Using execution controls on page 3-7
- Working with the Debug menu on page 3-10
- Controlling debugging on page 3-14.
3.1 Submitting commands

You can submit commands to RealView Debugger to control debugging behavior in several ways, for example by choosing from the Debug menu, or by clicking on a control on the Actions toolbar, or by typing a command-line instruction at the prompt.

If an application is currently executing, RealView Debugger uses a command queue to handle those commands that cannot be executed immediately. Through this mechanism, commands build up on the queue and are then executed when resources become available. Commands are never executed out of order.

If a command is currently executing and you request another action, the new command is added to the queue and pends until it can be executed. A warning message is displayed, in the Output pane, to explain what is happening to the new command, for example:

```
> go
Command pended until execution stops. Use 'Cancel' to purge.
```

If the application is still running, any further commands are then added to the queue behind this pending command.

The following notes apply to the command queue:

- All commands are added to the queue if they cannot be executed immediately.
- RealView Debugger appends unknown commands, and so possibly invalid commands, to the command queue.
- Breakpoints are set where possible, otherwise these commands also pend until they can be executed.
- Known invalid commands, for example those that do not start with a letter, are not added to the queue.

Click the Cancel button on the Actions toolbar to clear, or purge, the most recent pending command.
3.2 Defining execution context

RealView Debugger enables you to define the current execution context, and to change this if required. You do this using:

- Code views
- Defining scope and context.

3.2.1 Code views

Use the File Editor pane to view source code during your debugging session. In the example shown in Figure 2-3 on page 2-9, the File Editor pane contains three tabs:

- the Dsm tab enables you to track program execution in the disassembly-level view
- the Src tab enables you to track program execution in the source-level view
- the file tab dhry_1.c shows the name of the current source file in the editing, or non-execution, view.

Click on the relevant tab to toggle between the different code views.

The Src tab acts like a button to display the current source if it is available. If the source is not available, RealView Debugger displays a No source message in the tab.

If you click on a tab, for example the Src tab, and the statement where the PC is located is in view, a red box is drawn around that statement to highlight it in the chosen code view.

3.2.2 Defining scope and context

RealView Debugger uses scope to determine the value of a symbol. Scope shows how RealView Debugger accesses variables and finds symbols in expressions. The scope determines the execution context and defines how local variables are accessed. Any symbol value available to a C or C++ program at the current PC is also available to RealView Debugger.

When your program is executing, the PC stores the address of the current execution point. By default, the scope is set when the PC changes. Loading an image sets the PC at the entry point using autoscope, that is the PC defines the scope. Autoscope is also used in an assembly language routine when you step into code that has no source information. In this case, RealView Debugger shows the last calling function that had valid source in the Src tab.
RealView Debugger uses a red box to highlight the location of the PC where this is visible in the selected code view. The PC is only visible in an execution tab, that is the **Src** or **Dsm** tab. However, if you force scope to a different location then this is identified by a filled blue arrow and the red box highlights the current context. This might not be the true location of the PC. See *Forcing scope* for details.

When RealView Debugger first loads an image, and assuming that you do not force scope, the File Editor pane contains tabs showing program execution (described in *Code views* on page 3-3):

- the **Src** tab shows the current context, that is the location of the PC at the entry point
- the **Dsm** tab displays disassembled code with intermixed C/C++ source lines and, if available, the location of the PC.

### Locating the Program Counter

You can locate the PC using the *Debug* menu:

- select **Debug** → **Execution Control** → **Show Line at PC** to display the current location of the PC in the Output pane
- select **Debug** → **Execution Control** → **Show Context of PC** to display the current context of the PC in the Output pane.

### Forcing scope

Scope is forced when it is not set by the PC. To force the scope:

1. Connect to your target and load an image, for example `dhrystone.axf`.
2. Select **Edit** → **Editing Controls** → **Show Line Numbers** to display line numbers. This is not necessary but might help you to follow the examples.
3. Click on the **Src** tab to view the source file `dhry_1.c`. The PC is at the entry point at line 78, marked with a red box.
4. Right-click at a location (line or address) in your execution view, for example line 149 in the source file `dhry_1.c`.
5. Select **Scope To Here** from the context menu. The forced scope is identified by a filled blue pointer at the chosen location. This moves the red box to highlight the current context.
6. Select **Debug → Execution Control → Show Context of PC** to see the location of the PC in the Output pane.

7. Click **High-level Step Into** to step into the program.

8. Select **Debug → Execution Control → Show Context of PC** to see the location of the PC.

To reset the PC to the entry point, either:

- Select **File → Set PC to Entry Point** from the Code window main menu.
- Click **Set PC to Entry** on the Actions toolbar.

If you reset the PC to the entry point, this issues a **RESTART** command but does not reset the values of variables, reset the **Stack Pointer (SP)**, or clear breakpoints.

**Setting disassembly format**

When working in disassembly view, you can temporarily choose the display format for your code view:

1. Connect to your target and load an image, for example dhrystone.axf.
2. Click on the **Dsm** tab.
3. Right-click on whitespace inside the File Editor window to see the **Disassembly View** context menu, shown in Figure 3-1.

![Figure 3-1 Disassembly View menu](image)

---

**Controlling Execution**

ARM DUI 0153E  Copyright © 2002-2004 ARM Limited. All rights reserved.  3-5
4. Select **Set Disassembly Format...** from the context menu to see the Disassembly Mode selection box, shown in Figure 3-2.

![Disassembly Mode selection box](image)

**Figure 3-2 Disassembly Mode selection box**

Use this selection box to specify how disassembled code is displayed. If you choose Auto-mode, the default format, RealView Debugger displays the code in a format specified by the image contents.

5. Select the required format and click **OK** to confirm your choice. Click **Cancel** to close the selection box without changing the display format.

You can use workspace settings to specify the format used for disassembly. For details, see the description of the **DEBUGGER** group (Disassembler settings) in Appendix A **Workspace Settings Reference**.
3.3 Using execution controls

The **Execution** group, on the Actions toolbar, contains buttons to control the execution of the image loaded to your debug target. This section describes how to access execution controls:

- **Using the Execution group**
- **Using shortcuts** on page 3-9.

**Note**

In the following examples, loading the image `dhrystone.axf` places the PC at the entry point and not at the start of `main()`. RealView Debugger indicates the current context by highlighting the location of the PC with a red box. Any subsequent stepping instructions are based on this starting point.

3.3.1 Using the Execution group

The Execution group contains:

- **Go**
  
  Click this button to start from the current location of the PC and run until the program:
  - ends
  - encounters an error condition
  - reaches a breakpoint
  - reaches a halt condition.

  This option can also be used to resume program execution after stopping.

- **High-level Step Into**
  
  Click this button to step by lines of source code until the PC is located in another function or context. Normally, click this button to step up by one call level, but, depending on the function stepped into, it might cause the program to execute until it reaches source code.

  If the source line makes a function call, RealView Debugger steps into the function, unless there is no source code available for this function.

  To see an example using this option:

  1. Connect to your target and load an image, for example `dhrystone.axf`.
  2. Click on the **Src** tab to view the source file `dhry_1.c`.
  3. Select **Edit → Editing Controls → Show Line Numbers** to display line numbers.

     This is not necessary but might help you to follow the examples.
4. Display line 78.
5. Set a simple breakpoint by double-clicking on the line number in the left margin.
6. View the Output pane message:
   bi \DHRY_1\#78:5
7. Click Go and the program begins execution and runs up to the breakpoint.
   The Output pane shows where execution stops.
8. Click High-level Step Into once. A red box shows the location of the PC at line 91.
9. Click High-level Step Into several times to move through the lines of source code.

In this example, RealView Debugger completes two high-level steps at the start. Therefore, the first high-level step takes the PC from the entry point to main(), and the second steps to after the function prolog.

--- Note ---
You can get to main() using Go (as above) or a single step. The single step executes (from the high-level code) up to main().

---

High-level Step Over

Click this button to step by lines of source code over function calls.

If the source line makes a call to a function, RealView Debugger executes the function completely before stopping, assuming that there is no stopping condition in the function call, for example a breakpoint.

Low-level Step Into

Click this button to step by instructions into functions. RealView Debugger executes the assembly language instruction at the current location of the PC.

Low-level Step Over

Click this button to step by instructions over function calls. RealView Debugger executes the assembly instruction at the current location of the PC.

If the assembly instruction makes a call to a function, RealView Debugger executes the function completely before stopping, assuming that there is no stopping condition in the function call, for example a breakpoint.
Go until Return
Click this button to start from the current PC in the current function and to run until it returns to the calling function.

Go to Cursor
Click this button to start from the current location of the PC and to run until it reaches a temporary breakpoint, defined by the cursor position.

Stop Execution
Click this button to stop the program currently executing on the target processor.
If you are using semihosting with RealView ARMulator® ISS, you cannot use the Stop Execution button during the semihosting input.

Note
In source code, repeated uses of low-level step commands might be necessary to complete execution if the source line contains multiple ARM® instructions. Low-level step instructions, Low-level Step Into or Low-level Step Over, complete one assembly instruction at a time. This means that a call to a subroutine is treated as one instruction if you execute a step over.

3.3.2 Using shortcuts
The controls in the Execution group are independent of the current code view, that is clicking a button carries out the specified action regardless of whether you are in source-level view (the Src tab) or disassembly-level view (the Dsm tab). The Status line at the bottom of the Code window gives a description of the action available from a button.

Keyboard shortcuts are available, shown in the Execution Control submenu in the Debug menu, depending on the current code view. These are summarized in Table 3.1.

<table>
<thead>
<tr>
<th>Code view</th>
<th>Function key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>F10</td>
<td>Step by line over functions</td>
</tr>
<tr>
<td>Source</td>
<td>F11</td>
<td>Step by line into functions</td>
</tr>
<tr>
<td>Disassembly</td>
<td>F10</td>
<td>Step by instruction over functions</td>
</tr>
<tr>
<td>Disassembly</td>
<td>F11</td>
<td>Step by instruction into functions</td>
</tr>
</tbody>
</table>
3.4 Working with the Debug menu

Debugging your application programs relies on being able to control the execution of your code on the debug target. You must then be able to examine the contents of memory, registers, or variables, possibly continue execution one instruction at a time, or specify other actions to examine in detail the execution history. The Code window main menu includes the **Debug** menu, shown in Figure 3-3.

![Figure 3-3 Debug menu](image)

This section describes the **Debug** menu providing a starting point for many debugging operations:

- *Using the Debug menu*
- *Using the Execution Control menu* on page 3-11.

3.4.1 Using the Debug menu

The **Debug** menu offers the options:

**Execution Control**

This enables you to control how your program executes. You can run a series of instructions, step through instructions one at a time, or specify conditions that must be met to stop execution.

For details on using this option in your debugging session see *Using the Execution Control menu* on page 3-11.

**Simple Breakpoints**

This provides access to the **Simple Breakpoints** menu to use breakpoints in your code.

**Complex Breakpoints**

This provides access to the **Complex Breakpoints** menu to use breakpoints in your code. You must use a debug target that supports these breakpoints.
Tracepoints

This enables you to set tracepoints so that trace data can be captured.

See the chapter describing tracing in RealView Debugger v1.7 Extensions User Guide for full details.

Memory/Register Operations

This enables you to examine memory and register contents during execution and to edit these interactively. In this way you have complete control over the way your application program runs.

Include Commands from File...

This enables you to store debugger commands in the form of a script file and then run the file to automate the debugging process.

Set Source Search Path...

Use this to display the Project Properties windows to enable RealView Debugger to find sources where the compiler or assembler does not pass source paths (see Searching for source files on page 3-17 for details).

Add/Edit Debugger Macros...

This enables you to use the RealView Debugger macro facility to create macros containing complex procedures. You can attach a macro to a breakpoint or run it as a command.

3.4.2 Using the Execution Control menu

The Debug menu enables you to define the execution path using the Execution Control menu, shown in Figure 3-4. In most cases, you use the menu options in combination with other debugging tools such as breakpoints.

Figure 3-4 Execution Control menu
The **Execution Control** menu options are grouped according to their impact on the execution path:

**Reset Target Processor**

Performs a processor reset operation on the current connection.

The reset might be hard or soft depending on the processor type, see your processor hardware documentation for details.

Simulating a processor reset does not reset variables to their defaults because memory is not re-initialized. The PC is reset.

**Go (Start Execution)**

With an image loaded to the target processor, select this option to run the program, starting from the current location of the PC.

**Go to Cursor**

With an image loaded, you can scroll down the listing and position the cursor on a specific line. Select **Go to Cursor** to execute the program up to the temporary breakpoint at the cursor, assuming that no halting condition occurs first.

If you select the **Src** tab, the line marked by the cursor is enclosed in a red box indicating the position of the PC.

Click **Go**, or use a **Step** control, to resume execution.

**Go until Return**

Select this option to run from the current PC until control returns from the current function.

Selecting this option stops execution at the assembler instruction immediately after the return, and not the next statement. If you select the **Src** tab, this has the effect that the red box indicating the position of the PC might be still located at the function call.

**Step Into**

Steps execution, by lines of source code or assembler instructions, into functions. This behavior depends on the current code view, that is whether you have selected the **Src** tab or the **Dsm** tab.

**Step Over (next)**

Steps execution, by lines of source code or assembler instructions, over functions. This behavior depends on the current code view, that is whether you have selected the **Src** tab or the **Dsm** tab.
Step Until Condition...
Displays a prompt where you can specify a condition, in the form of an expression. When the condition is met, that is the condition is nonzero, execution halts.
Click OK to run in single steps from the current location of the PC. RealView Debugger checks the condition after each step.
Using this option causes execution to slow down and might result in errors due to timing issues.

Stop Execution
Select this option to stop the program currently executing on the target processor.

Show Line at PC
Select this option to report the current module and procedure scope.

Show Context of PC
Select this option to report the current context showing the current root, module, procedure, and line.

Toggle Source/Disassembly
Select this option to toggle between the source-level view and the disassembly-level view in the File Editor pane.
3.5 Controlling debugging

The Debug menu includes options that give you control over debugger operations:

- *Using include files*
- *Searching for source files* on page 3-17.

3.5.1 Using include files

RealView Debugger enables you to use include files to enter commands and so carry out debugging tasks without user intervention. The commands are actioned as though they are being entered from the keyboard.

During your debugging session, you can create a log file of all the commands you enter. This file can then be used as the basis of a command file or a macro. By default, log files have the extension `.log` or `.inc`, but you can use any extension for writing.

Another log file, called the `STDIOlog` log file, enables you to keep a record of debuggee output only, that is messages from the target. This might be useful for controlling debugging by running scripts without using the RealView Debugger user interface. By default, these files have the extension `.log`, but you can use any extension for writing.

In addition to the log files, you can also create a journal of a debugging session. The session journal file you create contains all information including your commands, debugger output and any messages displayed in the Output pane. By default, journal files have the extension `.jou`, but you can use any extension for writing.

In summary:

- log files record commands you enter and messages from the debugger
- STDIO log files record messages from the debuggee only
- journal files record commands you enter and messages from the debuggee.

The following sections describe how to manage your debugging session using include files such as log or journal files, or scripts you create yourself:

- *Output buffering*
- *Creating log and journal files* on page 3-15
- *Closing log and journal files* on page 3-16
- *Including files* on page 3-16.

Output buffering

In the current release of RealView Debugger, output to the log files and journal files is unbuffered. This means that all lines are immediately flushed to the specified file. To change this, so that output to a file is buffered, set the `JOULOG_UNBUF` environment variable.
Creating log and journal files

RealView Debugger writes log and journal output to a file saved in a specified location. If the file does not exist, RealView Debugger creates it. Where a file exists, RealView Debugger gives you the option to add new entries to the file, or to overwrite the current contents.

To create an output file, or to open an existing file for writing:

1. Select File → New → Log File... to display the Select File to Log to dialog box where the file can be located.
2. Specify the pathname of the new log file, or locate a file created previously, for example \home\my_user_name\my_log.log.
3. Click Save to confirm the settings and close the dialog box.
4. If the specified log file already exists, RealView Debugger displays the File Exists prompt.
   This gives you the option to append or replace. Click:
   • Yes to append new commands to those already saved in the file
   • No to replace, or overwrite, any commands already saved in the file
   • Cancel to close the prompt and discontinue the log file access.

Output is now recorded automatically in the specified file.

RealView Debugger shows that it is recording using the status display area at the bottom of the Code window, shown in Figure 3-5.

Using log and journal files at start-up

You can start RealView Debugger and open a log or journal file for writing. Do this from MS-DOS, or from a Command Prompt window, or create a desktop shortcut, for example:

rvdebug.exe -s "C:\rvd\test_files\STDIO_tst_file.log"
rvdebug.exe -log "C:\rvd\test_files\Log_tst_file.log"

If the file does not exist, RealView Debugger creates it. Where the file exists, RealView Debugger overwrites the current contents, without displaying a warning message.
When RealView Debugger starts to write to the log file, it records the filename as the first entry, for example:

;;;LOG FILE: C:\rvd\test_files\my_log.log

**Closing log and journal files**

If you are recording a log, or journal file and you try to start a new recording, RealView Debugger gives you the option to close the current file so that a new file can be used.

To close a log or journal file, select **File → Close Logs/Journals...**. This displays a list selection box, shown in Figure 3-6, where you can specify which file, or files, to close.

![Figure 3-6 Close log and journal files selection box](image)

Each entry has an associated check box that is ticked by default. Select a check box to unselect a file. The list selection box contains:

**OK**

Click this button to close selected files and then close the selection box.

**Cancel**

Click this button to leave all files open and then close the selection box. Using **Cancel** ignores the status of any check boxes in the list.

**Help**

Click this button to display the online help for this selection box.

Use the File Editor pane, or a text editor of your choosing, to view the contents of your log and journal files. You can then edit the commands shown in a log file to create an include file for use as a command file or as a macro.

**Including files**

Use a text editor to create a file of commands that can then be submitted to RealView Debugger to control a debugging session. To use an include file:

1. Select **Debug → Include Commands from File...**, from the Code window, to display the Select File to Include Commands from dialog box.
2. Locate the file where your debugger commands are stored. By default, RealView Debugger looks for a .inc or a .log file.

3. Click **Open** to load the file and execute the commands stored there.

You can also start RealView Debugger with an include file, for example:

```
rvdebug.exe -inc "C:\rvd\test_files\my_cmds_file.inc"
```

### 3.5.2 Searching for source files

By default RealView Debugger searches for application source file paths according to information contained in the image. If no paths are provided in the image, RealView Debugger uses the list of directories specified in the project settings. If this fails, or where no search path has been specified, RealView Debugger looks in the current working directory for the source file or files.

If you have loaded an image to a target, or where you have opened a user-defined project, you can specify search paths using the Project Properties window. Any search that you define is then saved with the project settings and additionally used to locate source files.

In the following example, the image has been built in one directory and then moved to another location for debugging. This means that RealView Debugger cannot locate source files directly from information contained in the image:

1. Ensure that the Project Properties window is not open. When searching for source files, RealView Debugger updates the project properties as necessary. This fails if the window is already open.

2. Close any files that are open in the File Editor so that they are not found, and included, in the search mechanism.

3. Connect to your target and load an image, for example **dhry_thr_demo.axf**.

   RealView Debugger creates an in-memory auto-project to use for this session, that is **dhry_thr_demo.axf.apr**.

4. RealView Debugger cannot locate a source file and so displays the source search prompt, shown in Figure 3-7 on page 3-18.
If you close the prompt, for example by clicking Cancel, RealView Debugger warns you that it cannot locate the source file for the new image. The project properties are unchanged.

You can change the search rules that RealView Debugger uses. There are two ways to do this:

- **Autoconfiguring search rules**
- **Manually configuring search rules** on page 3-19.

**Autoconfiguring search rules**

Use the source search prompt, shown in Figure 3-7, to autoconfigure the search rules:

1. Either:
   - Edit the pathname shown in the prompt.
   - Click the directory button and use <Select file...> to locate the required file.
2. Click Retry.

If RealView Debugger fails to locate the file, the source search prompt remains for you to try again. If RealView Debugger succeeds, the source search prompt closes automatically.

---

**Note**

If the required file is already listed when you click the directory button, select it to update the project properties. The source search prompt then closes automatically.

---

Pathnames that you add this way automatically update the project properties for the active project. Where your project is an in-memory auto-project, RealView Debugger also saves the project settings file, for example dhry_thr_demo.axf.apr, in the same location as the original image.
Absolute pathnames are used to define the new search rules when you configure them in this way. However, if you modify the source search path yourself, pathnames become relative (see Manually configuring search rules for details).

Manually configuring search rules

If you have an open project, you can specify search paths using the Project Properties window. Any search that you define is then saved with the project settings and additionally used to locate source files.

To configure the search rules manually:

1. Select **Project → Project Properties...** to display the Project Properties window, shown in Figure 3-8.

![Figure 3-8 Search rules in the Project Properties window](image)

For more details on other entries in this window see *Source path mappings* on page 3-20.

2. Right-click on **Source_search** and use **Edit as Directory Name...** to specify the search path to locate the missing source file.

3. Click **File → Save and Close** to confirm your choice and close the Project Properties window.

Do not leave the Project Properties window open when you are working with the debugger. When searching for source files, RealView Debugger updates the project properties as necessary. This fails if the window is already open.
Note

Relative pathnames are used to define the new search rules when you configure them in this way.

Source path mappings

When you use the directory button to locate the source file, using the prompt shown in Figure 3-7 on page 3-18, RealView Debugger creates a mapping between the original file and the new location. This mapping is then applied to subsequent file searches so that the debugger can locate files automatically that have the same mapping.

Pathname remappings are stored in the project settings so that they can be used in the next session.

Note

Where your project is an in-memory auto-project, RealView Debugger automatically saves the auto-project if you set up a new pathname mapping.

See the chapter describing customizing projects in RealView Debugger v1.7 Project Management User Guide for full details on working with these entries.
Chapter 4
Working with Breakpoints

This chapter explains the different types of breakpoints supported by RealView® Debugger, describes the options for setting breakpoints, and explains how to manage breakpoints during your debugging session. It includes the following sections:

- Breakpoints in RealView Debugger on page 4-2
- Setting breakpoints quickly on page 4-7
- Using simple breakpoints on page 4-13
- Setting conditional breakpoints on page 4-22
- Setting hardware breakpoints on page 4-28
- Using complex breakpoints on page 4-34
- Using the Break/Tracepoints pane on page 4-37
- Disabling and clearing breakpoints on page 4-43
- Saving breakpoints as favorites on page 4-45.
4.1 Breakpoints in RealView Debugger

Breakpoints are specified locations where execution must stop. The breakpoint can be triggered by:

- execution reaching the specified address
- data reads or writes at a specified address or address range
- breakpoint qualifiers passing specified test criteria
- data values at the specified location, in the current context, becoming equal to a particular value or range.

When a breakpoint triggers, RealView Debugger can carry out higher level requests. For example you can:

- attach macros to breakpoints
- update windows or files
- change the behavior of your application program.

You can also continue execution of your application program after RealView Debugger completes the specified operations.

This section describes:

- Breakpoint types
- Running and Halted System Debug on page 4-4
- Using hardware breakpoints on page 4-4
- Using breakpoints with RealView ICE on page 4-6.

4.1.1 Breakpoint types

RealView Debugger enables you to use different types of breakpoint. What is available, and the type of breakpoint set, depends on the:

- memory map, if enabled
- hardware support provided by your target processor
- target vehicle used to maintain the connection.

In RealView Debugger there are two types of breakpoint:

Simple

Simple breakpoints enable you to test address-specific data values. These breakpoints can be either hardware or software breakpoints.

When debugging code in ROM, RealView Debugger implements a simple hardware breakpoint by default. However, this depends on the hardware characteristics of your target processor.
Simple breakpoints are supported by all ARM® processors.

**Complex**

Complex breakpoints use advanced hardware support on your target processor, or as implemented by your simulator software. Where supported by your debug target, complex breakpoints might be data-dependent or take advantage of range functionality, for example two breakpoints can be coupled together.

Check your hardware characteristics, and your vendor-supplied documentation, to determine the level of support for complex breakpoints (see [Viewing your hardware characteristics](#) on page 4-5 for details).

By default, the memory map is disabled when you first start to use RealView Debugger. This means that all memory is designated read/write, that is RAM. In this case, RealView Debugger always sets a software breakpoint by default.

However, if the software breakpoint fails, for example, for a Flash-based target, RealView Debugger sets a hardware breakpoint instead. When a hardware breakpoint is used instead of a software breakpoint, RealView Debugger gives a warning about the substitution so that you can map memory as required.

Where the memory map is enabled, RealView Debugger sets a breakpoint based on the access rule for the memory at the chosen location:

- A hardware breakpoint is set for areas of no memory (NOM), Auto, read-only (ROM), or Flash.
- If the memory is write-only (WOM), or where an error is detected, RealView Debugger gives a warning and displays the Set Address/Data Break/Tracepoint dialog box for you to specify the breakpoint details.

For more details see Chapter 5 *Memory Mapping*.

RealView Debugger uses breakpoints to temporarily halt execution as you step through your application. By default, a software breakpoint is set, unless you have enabled memory mapping to define memory as read-only (ROM) or Flash. When a hardware breakpoint is used instead of a software breakpoint, RealView Debugger displays a message to warn you of the change.

For more details see *Using execution controls* on page 3-7.
4.1.2 Running and Halted System Debug

RealView Debugger supports different debugging modes:

**Halted System Debug**

Halted System Debug (HSD) means that you can only debug a target when it is not running. This means that you must stop your debug target before carrying out any analysis of your system. This debugging mode places no demands on the application running on the target.

HSD is always available as part of the RealView Debugger base product.

**Running System Debug**

Running System Debug (RSD) means that you can debug a target when it is running. This means that you do not have to stop your debug target before carrying out any analysis of your system.

RSD is only available where supported by your debug target. It relies on having RealView Debugger RTOS extensions installed and is not provided as part of the base product.

RSD gives access to the application using a Debug Agent (DA) that resides on the target and is typically considered to be part of the RTOS. The Debug Agent is scheduled along with other tasks in the system.

Where RSD is supported, RealView Debugger enables you to switch seamlessly between RSD and HSD mode using GUI controls or CLI commands.

For full details on using RTOS extensions and running in RSD mode, see the chapter describing RTOS support in *RealView Debugger v1.7 Extensions User Guide*.

4.1.3 Using hardware breakpoints

Setting a software breakpoint requires that the debugger changes executable instructions, so this is only possible for code stored in RAM. Where instructions are in ROM, you must set hardware breakpoints.

The number of hardware breakpoints available depends on your debug target. If RealView Debugger cannot set breakpoints, a warning message is displayed and rapid instruction step is used for high-level stepping. Again, a warning message is displayed to explain the type of step being used.
Be aware of the following when working with hardware breakpoints with *RealView ARMulator® ISS* (RVISS):

- Watchpoints are available. These are called hardware breakpoints in RealView Debugger and can be accessed through the **Debug** menu. These data access breakpoints are implemented using a memory hook.
- Hardware breakpoints can use address ranges, data values, and data value range tests. They can also include size tests, mode tests, and pass counts.
- Hardware breakpoints can be chained to form complex tests.

To use hardware breakpoints, your debug target must include support for such breakpoints. Even where this support is available, your target might be limited in the number it can support at one time. RealView Debugger menu options related to hardware breakpoints are grayed out if your target cannot support them or if no more are available.

**Viewing your hardware characteristics**

To see your hardware support for breakpoints select **Debug → Complex Breakpoints → Show Break Capabilities of HW...** from the default Code window main menu. This displays an information box describing the support available for your target processor, shown in Figure 4-1.

![Figure 4-1 Example hardware break characteristics](image)

What is shown in this details display depends on the target processor and the target vehicle used to make the connection. Figure 4-1 shows the details for an ARM940T using Multi-ICE®.
Note

RealView Debugger reserves one breakpoint unit for internal use and so this might not be available to you. You are warned if you try to set a hardware breakpoint when the limit is reached.

4.1.4 Using breakpoints with RealView ICE

If you are using RealView ICE, RealView Debugger sets breakpoints differently. For full details on debugging with RealView ICE, see RealView ICE User Guide.
4.2 Setting breakpoints quickly

You can set a breakpoint directly from the default Code window, depending on your current code view. This is useful to set a quick test point during a debugging session. This section describes:

- Quick breakpoints in source-level view
- Quick breakpoints in disassembly-level view on page 4-8
- Viewing breakpoints in your code view on page 4-10
- Clearing breakpoints quickly on page 4-11
- Setting breakpoints in other ways on page 4-11.

In these examples, Text Coloring is enabled by default and line numbering is turned on by selecting Edit → Editing Controls → Show Line Numbers.

4.2.1 Quick breakpoints in source-level view

Double-click in the margin, that is the gray area to the left of a line, to set a breakpoint quickly, shown in Figure 4-2. You can also double-click on the line number if this is visible.

![Figure 4-2 Setting a breakpoint quickly on a line](image)

This sets a simple breakpoint marked by a red disc in the margin.

If you want to set a breakpoint on a multi-statement line, for example on a for... loop, right-click on the statement to display the context menu shown in Figure 4-3 on page 4-8, and select Set Break on Statement.
Working with Breakpoints

If you want to set a breakpoint on a symbol, right-click on the symbol to display the context menu, shown in Figure 4-3, and select Set Break On. You can also use this option to set a breakpoint in the body of a function by right-clicking on the call to the function.

Setting a breakpoint updates the Break/Tracepoints pane, if it is visible, and the Output pane shows the breakpoint command, for example:

```
   bi \DHRY_1\ #146:3
```

Setting breakpoints in source-level view inserts a software instruction breakpoint by default. This is set using a BREAKINSTRUCTION command. RealView Debugger attempts to set a software breakpoint where the code is not in ROM or Flash.

If your code is in ROM or Flash, RealView Debugger sets a hardware breakpoint where one is available. An error message is displayed if no such breakpoint is available.

See Using the Break/Tracepoints pane on page 4-37 for details on viewing breakpoints in the Break/Tracepoints pane.

Note

For details on the BREAKINSTRUCTION command see RealView Debugger v1.7 Command Line Reference Guide.

4.2.2 Quick breakpoints in disassembly-level view

Double-click on the required instruction, on the Dsm tab, to set a breakpoint quickly, shown in Figure 4-4 on page 4-9.
Figure 4-4 Setting a breakpoint quickly on an instruction

To set a breakpoint on the destination of a branch instruction:

1. Locate the required instruction in the Dsm tab.
2. Right-click to display the context menu, shown in Figure 4-5.
3. Select Set Instr Break.

4. The Break/Tracepoints pane is updated with the new breakpoint (if visible) and the Output pane shows the breakpoint command:

   ```
   bi 0x8064
   ```

As with the source-level view, RealView Debugger sets a software or hardware breakpoint depending on where your program is stored and what breakpoints are available.
4.2.3 Viewing breakpoints in your code view

Breakpoints are marked in the source-level and disassembly-level view in the margin, at the left side of the window, using color-coded discs or icons:

**Red**  
This symbol shows the position of an enabled breakpoint. A second breakpoint cannot be set at the same location as an existing breakpoint.

**Yellow**  
If you set a conditional breakpoint, that is one that stops execution when certain conditions are met, the marker is a disc filled with yellow.

**White**  
Where a breakpoint has been set previously and then disabled, it is marked by a white disc. If the breakpoint is re-enabled the disc changes color as appropriate for the type of breakpoint.

If you have set multiple breakpoint units, for example on a source line containing multiple statements or on inline functions, then disabling the first breakpoint unit changes the marker disc, shown in Figure 4-6. If you disable the second breakpoint unit on the line, the marker remains.

If you try to set a breakpoint on a non-executable line, RealView Debugger looks for the first executable line immediately following and places the breakpoint there. If the lines preceding the breakpointed instruction are comments, declarations, or other non-executable code, they are marked with black, downward pointing arrows. Lines marked in this way are regarded as part of the breakpoint. You cannot place two unconditional breakpoints on the same line or on lines marked by the downward pointing arrows.

---

See *Using the Break/Tracepoints pane* on page 4-37 for details on viewing breakpoints in the Break/Tracepoints pane.

Figure 4-6 Working with breakpoint units
Other types of breakpoints

If you have set tracepoints or thread breakpoints, these are shown as different types of marker. For example, if you define a trigger that instructs the debugger to stop collecting trace information and display the results, this is shown as a green right-facing arrow in the current code view. See RealView Debugger v1.7 Extensions User Guide for details of using these features.

4.2.4 Clearing breakpoints quickly

With a breakpoint visible in your current code view, you can clear it quickly by double-clicking on the marker disc in the margin. This removes the breakpoint you set.

Note
Where you have set multiple breakpoint units, clearing a breakpoint this way removes only the first breakpoint unit.

4.2.5 Setting breakpoints in other ways

RealView Debugger includes other ways to set new breakpoints and manipulate existing ones:

- Use drag-and-drop to create a simple breakpoint in the Break/Tracepoints pane based on a program item, for example highlight a source code function in the File Editor pane and then drag it (using your mouse) and drop it into the Break/Tracepoints pane (see Using the Break/Tracepoints pane on page 4-37 for details).

- Set a breakpoint on a memory location in the Memory pane. The type of breakpoint offered depends on the type of memory at the chosen location, for example RAM, ROM, or Flash.

- Set a hardware breakpoint at an address of a symbol using the Stack pane. However, this is not recommended if execution runs past the end of a function return call because as soon as you exit the function the stack value is no longer meaningful.

Use the Watch pane to track a specific symbol continuously because this recomputes the stack location dynamically and so tracks each invocation of the function.

You can use local variables within the conditional part of a breakpoint because the stack value is computed correctly each time the breakpoint condition is evaluated.

- Set a conditional breakpoint on a value shown in the Watch pane.
Working with Breakpoints

- Set a conditional breakpoint on a variable shown in the Call Stack pane.
- Set a simple breakpoint at the current cursor position using Debug → Simple Breakpoints → Toggle Break at Cursor.
- You can use the Function List dialog box to set a breakpoint on a chosen function, defined as a location in the image:
  1. Select Find → Function List... from the Code window main menu.
  2. Highlight a function in the display list.
  3. Click Break to set a breakpoint.
- A cut-down version of the Function browser is available to set a breakpoint by function name:
  1. Click on the location of your breakpoint in your code view.
  2. Select Debug → Simple Breakpoints → Set from Function/Label list... to display the Function Breakpoint/Profile Selector dialog box.
  3. Select the function by clicking the check box.
  4. Click Set.

Because the browser is used only to make a selection, there are no controls for debugging operations.

The Function Breakpoint/Profile Selector does not provide a record of breakpoints already set, that is, when you next open this dialog box existing breakpoints are not checked.
4.3 Using simple breakpoints

The Debug menu provides a range of breakpoint options to use during your debugging session. Select Debug → Simple Breakpoints from the Code window main menu to display the Simple Breakpoints menu, shown in Figure 4-7.

![Simple Breakpoints menu](image)

This menu offers different breakpoint operations:

- Using the Set Address/Data Break/Tracepoint dialog box
- Setting simple conditional breakpoints on page 4-17
- Breakpoint operations on page 4-20
- Setting breakpoints from saved lists on page 4-20.

4.3.1 Using the Set Address/Data Break/Tracepoint dialog box

Select Debug → Simple Breakpoints → Address/Data... from the Code window main menu to display the Set Address/Data Break/Tracepoint dialog box, shown in Figure 4-8 on page 4-14.
The Set Address/Data Break/Tracepoint dialog box provides comprehensive facilities to enable you to specify new breakpoints in full. You can also use it to edit existing breakpoints.

The main interface components of the Set Address/Data Break/Tracepoint dialog box are:

**Location**  
Specifies the memory location where the new breakpoint is set. Click the drop-down arrow to the right of this field to choose from a list browser, or select from your personal favorites list, or select from a list of previously-used expressions. The options shown here depend on your debug target and connection.

Where supported by your target hardware, use the options from the right-arrow menu to qualify the location (see Using ranges and masks on page 4-31 for details).

This field is enabled if you select a suitable Break/Tracepoint Type and your current target supports the chosen type.
Value Match

Enter the data value that triggers the breakpoint. Click the drop-down arrow to the right of this field to choose from a list browser, or select from your personal favorites list, or select from a list of previously-used expressions.

If you use this with data breakpoints, this compares the data value that is read or written.

Where supported by your target hardware, use the options from the right-arrow menu to qualify the value match (see Using ranges and masks on page 4-31 for details).

This field is enabled if you select a suitable Break/Tracepoint Type and your current target supports the chosen type.

Class

A read-only field to show the type of breakpoint you set.

By default, RealView Debugger sets a Standard Breakpoint but the contents of this field change depending on your debugging environment and target configuration. For example, the Class field might show Tracepoint or Thread Breakpoint.

See the chapters describing tracing and RTOS support in RealView Debugger v1.7 Extensions User Guide for more details on breakpoint classes, and how to work with different types of breakpoint or tracepoint.

Break/Tracepoint Type

Enables you to select the type of breakpoint to set. On first opening the dialog box, the list shows the breakpoint types that are supported by your debug target.

If you select a Break/Tracepoint Type, the contents of the groups in this dialog box might change.

HW Support

This area is populated if you select a suitable Break/Tracepoint Type and your current target supports the chosen type.

Where your debug target supports breakpoint tests in hardware, they can be managed and edited using this group. If enabled, the display lists currently available tests.

See Setting hardware breakpoints on page 4-28 for details on using these controls.
Qualifiers  When setting a conditional breakpoint, you specify the condition that must be satisfied to trigger the breakpoint. Qualifiers are the tests that RealView Debugger carries out to trigger the breakpoint. Click New to display the New Qualifiers menu, shown in Figure 4-9, where you can define the test criteria.

Figure 4-9 Breakpoint New Qualifiers menu
See Setting conditional breakpoints on page 4-22 for details on using these controls.

Actions  When a conditional breakpoint triggers the usual action is to stop execution but you can specify one or more debugger actions that must be performed when execution stops. In addition, RealView Debugger can carry out the specified action and then execution can continue. This is useful when debugging complex applications without direct user intervention.

Click New to display the New Actions menu, shown in Figure 4-10.

Figure 4-10 Breakpoint New Actions menu
See Setting conditional breakpoints on page 4-22 for details on using these controls.

OK  Click OK to confirm the new breakpoint properties and close the dialog box.

Cancel  Click Cancel to close the dialog box and abandon the breakpoint setting.

Help  Click Help to get online help on the controls in this dialog box.

Note  Depending on the Break/Tracepoint Type you select, the Location or Value Match fields might be unavailable. In this case, the field is grayed out. Also, in this example, the Class field is read-only and the drop-down arrow is unavailable.
4.3.2 Setting simple conditional breakpoints

The Simple Breakpoints menu, shown in Figure 4-7 on page 4-13, also enables you to set simple breakpoints quickly from the default Code window:

Set from Function/Label list...

Enables you to set a breakpoint on any number of the function names and labels in your image.

The Function Breakpoint/Profile Selector dialog box does not provide a record of breakpoints already set, that is, when you next open this dialog box existing breakpoints are not checked.

Simple Break if X...

Displays the dialog box, shown in Figure 4-11, enabling you to specify an expression that evaluates to an address.

![Figure 4-11 Simple Break if X dialog box](image)

The Breakpoint Type controls the type of memory, program, or data, and the type of access that stops execution. In this case, this shows SW Instr as the given type. This field is set to read-only where this is the only type of breakpoint supported by your debug target.

If your target supports hardware breakpoints, click on the drop-down arrow to display a list of the available types.

Simple Break if X, N times...

This option is similar to the previous option except that now you can specify how many times execution must arrive at the specified address before the breakpoint triggers. Select this option to display a dialog box where you can specify an address for a breakpoint, shown in Figure 4-12.

![Figure 4-12 Simple Break if X, N times dialog box](image)
The additional field, \textit{After \_ times}, enables you to specify the number of times execution must arrive at the specified address to trigger the breakpoint, for example, when \texttt{Proc_4} has been executed 150 times.

\begin{verse}
\textbf{Note}
\end{verse}

If you are using a debug target that supports it, the pass count can be made in hardware.

\section*{Simple Break if X, when Y is True...}

Displays a dialog box where you can specify an address for a breakpoint, shown in Figure 4-13.

![Figure 4-13 Simple Break if X, when Y is True dialog box](image)

The additional field, \textit{When expression is True}, enables you to specify an expression (given in C format) that must be true when execution arrives at the specified address for the breakpoint to be triggered.

\section*{Named...}

When working with a user-defined project or an auto-project, you can specify named breakpoints that are saved in the project and so are available during your debugging session.

\begin{verse}
\textbf{Note}
\end{verse}

The list of named breakpoints is maintained as part of the project SETTINGS group. See the chapter describing customizing projects in \textit{RealView Debugger v1.7 Project Management User Guide} for details on how to set up these breakpoints.

If you have specified named breakpoints, the \textbf{Named...} option is enabled. Click here to select one or more breakpoints from the predefined list, shown in Figure 4-14 on page 4-19.
Figure 4-14 Named breakpoints list

Figure 4-14 shows the named breakpoints specified in the open project. Select the check box for the breakpoint(s) that you want to set and then click **OK** to close the list box and set the breakpoint(s). An item in the list that is grayed means that a symbol is specified for the breakpoint but this is not loaded.

**Processor Events...**

If supported by your debug target, RealView Debugger maintains a list of processor events that automatically trigger a breakpoint in any application program. During your debugging session you can examine this list and select, or deselect, halting events. Select this option to display a list selection box, shown in Figure 4-15.

Figure 4-15 Global Breakpoints list selection box

The list box shows processor events that stop execution. An event is enabled when the associated check box contains a tick. Click on a check box to enable, or disable, a chosen event.

These are global breakpoints, that is they apply to processor events and not addresses.
4.3.3 Breakpoint operations

The Simple Breakpoints menu enables you to complete certain breakpoint operations:

**Toggle Break at Cursor**

Sets a simple breakpoint at the address defined by the position of the cursor in your code view. If a breakpoint already exists at this address use this option to clear it.

**Enable/Disable Break at Cursor**

Enables a breakpoint at the address defined by the position of the cursor in your code view. If there is no disabled breakpoint at this position, select this option to create a new breakpoint. If an enabled breakpoint already exists at this address select this option to disable it.

**Clear All Break/Tracepoints**

Clears all breakpoints set on the current target.

See Disabling and clearing breakpoints on page 4-43 for more details of disabling and clearing breakpoints and tracepoints.

4.3.4 Setting breakpoints from saved lists

Your personal history file, exphist.sav, is saved in your RealView Debugger home directory and is updated when you close down at the end of your session. It contains a snapshot of the current breakpoints across all your debug targets. The items in this list accumulate during this, and previous, debugging sessions.

To see the current breakpoint history, select Debug → Simple Breakpoints → Break/Tracepoint History... from the main menu.

Breakpoints are only added to the history list if they are set using breakpoint dialog boxes, for example the Set Address/Data Break/Tracepoint or the Simple Break if X dialog box. If you set a breakpoint in another way, for example using a CLI command, this is not added to the list. To force this type of breakpoint to be added to the history list:

1. Set the required breakpoint, for example using the CLI command.

2. Select View → Pane Views → Break/Tracepoints Pane from the Code window main menu to display the Break/Tracepoints pane, shown in Figure 4-30 on page 4-38.

3. Right-click anywhere along the entry (not on the check box) to display the Break/Tracepoints context menu, shown in Figure 4-33 on page 4-40.
4. Select **Edit Break/Tracepoint...** to display the Set Address/Data Break/Tracepoint dialog box.

5. Click **OK** to close the dialog box without changing the breakpoint details.

See *Saving breakpoints as favorites* on page 4-45 for details of creating breakpoint favorites and adding existing breakpoints to your personal favorites list.

——— **Note** ———

If you are using RealView Debugger on non-Windows platforms, the history file is only created if you create and save a favorite, for example a breakpoint or watchpoint. See Appendix B *RealView Debugger on Sun Solaris and Red Hat Linux* for details.
4.4 Setting conditional breakpoints

When setting conditional breakpoints in your application program you can specify actions and qualifiers that control how RealView Debugger handles the breakpoint:

**Qualifiers**
Test conditions that must be satisfied to trigger the breakpoint, for example testing a variable for a given value, or executing a function a set number of times, or successfully running a macro.

**Actions**
These are debugger actions completed when the breakpoint triggers, for example displaying a message or updating a window.

**Continuation state**
Specify the execution behavior immediately following completion of the actions, that is the program can stop or continue.

Conditional breakpoints can be very intrusive because RealView Debugger takes control when the breakpoint triggers. The specified qualifiers are checked and then, if applicable, control is returned to the application.

When a conditional breakpoint triggers, there might be a discrepancy between the real state of your target and what is shown in the Code window. For example, the State field might show that the target has stopped when it is running. This is because the state of the target, as reported by RealView Debugger, is a snapshot of the target status at the time that the breakpoint triggered. Depending on the tasks on the system, the target state might have changed. Therefore, when you are using conditional breakpoints, remember that the state of the target depends on several factors, including:

- the speed of your debug target and the processor, or processors, it contains
- those instructions currently being executed on the target
- how much processing is required on the host to resolve the conditional breakpoint.

**Note**
If you are using RTOS extensions and running in RSD mode, see the chapter describing RTOS support in *RealView Debugger v1.7 Extensions User Guide* for more details on working with breakpoints.

This section describes how to manage actions and qualifiers in conditional breakpoints:

- *Managing qualifiers and actions* on page 4-23
- *Attaching macros to breakpoints* on page 4-26.
4.4.1 Managing qualifiers and actions

Qualifiers, actions, and the continuation state, are set up using the Set Address/Data Break/Tracepoint dialog box, shown in Figure 4-8 on page 4-14.

Using the New Qualifiers menu

Click the New button in the Qualifiers group to display the New Qualifiers menu, shown in Figure 4-9 on page 4-16. You can use this to specify qualifiers when you first create a breakpoint, or to add qualifiers to edit an existing breakpoint.

This menu enables you to select the qualifier that controls execution, that is to define the condition that must be satisfied to trigger the breakpoint:

SW Pass Count...
Use this to specify the number of times execution must arrive at the specified address before execution stops.

When Expression True...
Enables you to specify an expression that must evaluate to true to stop execution.

When Expression False...
Use this to specify an expression that must evaluate to false to stop execution.

User Macro...
Enables you to specify a macro that runs when execution stops. This brings up a dialog box where you supply the macro name and any arguments required to run it. See Attaching macros to breakpoints on page 4-26 for an example.

C++ Object...
Use this to specify a C++ this object to test. The Call Stack pane contains a This tab where you can view such objects.

Favorites...
Select this option to display your personal favorites list of breakpoint qualifiers. From here you can specify the required qualifier. See Saving breakpoints as favorites on page 4-45 for details of creating breakpoint favorites and adding qualifiers to the list.
Using the New Actions menu

Click the New button in the Actions group to display the New Actions menu, shown in Figure 4-10 on page 4-16. You can use this to specify breakpoint actions when you first create a breakpoint, or to add actions to edit an existing breakpoint. These actions are not actioned until the breakpoint qualifiers complete:

Update Window...
Displays a list selection box where you can choose which debugger windows are updated when execution stops. The list includes all windows and panes available from the default desktop. You can also redirect debugger output to a user window and this can be updated when execution stops.
You can only use this selection box to specify one window at a time. If you want to update several windows, repeat the operation to set up a windows list. The list order specifies the update order.

Update All Windows
Updates all desktop windows when the breakpoint triggers.

Update Sample...
Updates registered samples when execution stops. This is used as part of the graphing and visualization functions in RealView Debugger.

Note
This menu option is not available for visualization functions in this release. This option is available when a sampling variant is available, for example logging from breaks.

Breakpoint Timer
This option is enabled where your debug target supports cycle timing in hardware. The timer measures execution time from the point where the breakpoint triggers.

Message Output...
Displays a dialog box where you can enter a short text string for display when the breakpoint triggers.
By default this message appears in the Output pane but you can specify a window, for example, $250$Stop at convert proc. This sends the message Stop at convert proc to the specified window (see Attaching macros to breakpoints on page 4-26 for an example).
Favorites... Select this option to display your personal favorites list of breakpoint actions. From here you can specify the required action.

If you have used actions previously, these are displayed at the bottom of the New Actions menu.

Viewing qualifiers and actions

When you have set up actions and qualifiers they are displayed in the Qualifiers and Actions group display lists, shown in the example in Figure 4-16.

![Figure 4-16 Breakpoint actions and qualifiers](image)

The order of the qualifiers, in the Qualifiers group display list, defines the order they are tested to trigger the breakpoint. If a qualifier fails then subsequent conditions are not tested.

The order of the actions, in the Actions group display list, defines the order they are carried out when the breakpoint triggers. These actions do not execute until all breakpoint qualifiers complete successfully.

To manage the Qualifiers and Actions display lists:

- re-order the list by highlighting a qualifier, or action, and use the up, or down, button to reposition the specified entry in the list
• highlight a qualifier, or action, and click **Del** to delete the specified entry
• highlight a qualifier, or action, and click **Edit** to update it so changing the behavior.

### 4.4.2 Attaching macros to breakpoints

RealView Debugger includes a macro facility that enables you to create macros containing complex procedures that are then executed on your host workstation. You can attach a macro to a breakpoint so that it is executed when the breakpoint triggers. The macro can return values that determine whether program execution continues or stops.

RealView Debugger recognizes several predefined macros containing commonly used functions. These macros can also be attached to breakpoints. However, if you are attaching a macro that you create yourself, for example the `tutorial()` macro created in *Using macros* on page 9-8, then this must be open in the debugger.

To open a macro ready to attach it to a breakpoint:

- Select **Debug → Include Commands from File...** to display the Select File to Include Commands from dialog box.
- Highlight the macro file, for example `tutorial.inc`.
- Click **Open** to open it into the debugger.

To attach a macro to a breakpoint:

1. Select **Debug → Simple Breakpoints → Address/Data...** to display the Set Address/Data Break/Tracepoint dialog box.
2. Enter the location of the breakpoint, for example **0x8704**.
3. Click the **New** button in the Qualifiers group to display the **New Qualifiers** menu, shown in Figure 4-9 on page 4-16.
4. Select the option **User Macro...** to display the data entry prompt where you enter the macro name, shown in Figure 4-17 on page 4-27.
This predefined macro displays a message box to halt execution if the Yes key is pressed.

5. Click Set to confirm your entry.

6. The Set Address/Data Break/Tracepoint dialog box now contains the macro in the Qualifiers group for the conditional breakpoint, shown in Figure 4-18.

7. Click OK to confirm the breakpoint settings and close the dialog box.

8. Click Go to execute the program and trigger the breakpoint.

**Note**

Execution-type commands are not valid within a breakpoint macro. See Chapter 9 Working with Macros for full details.
4.5 Setting hardware breakpoints

Use hardware breakpoints to set data breaks, or where your code is stored in ROM. The facilities available depend on the current debug target, that is both the target processor and the target vehicle. Menu options related to hardware breakpoints are grayed out if your target cannot support them. Similarly, RealView Debugger displays a message if you select an option from a drop-down list box that is not supported by your debug target.

This section describes:

- Setting simple hardware breakpoints
- Using ranges and masks on page 4-31
- Setting hardware breakpoints on a DSP-based debug target on page 4-33.

4.5.1 Setting simple hardware breakpoints

To set hardware breakpoints, you must be connected to a debug target that supports these features, for example an ARM processor with EmbeddedICE® logic. The options shown depend on the target support for breakpoints in hardware.

Select **Debug → Simple Breakpoints → Address/Data...** to display the Set Address/Data Break/Tracepoint dialog box, shown in Figure 4-19.

![Figure 4-19 Set Address/Data Break/Tracepoint dialog box](image-url)
Select a hardware Break/Tracepoint Type to display entries specific to the hardware support for breakpoints available on the current target:

**HW Instr**
Sets or modifies an instruction address breakpoint. This type of breakpoint enables you to perform hardware tests or to compare data values.

**HW Read**
Sets or modifies a read breakpoint at the specified memory location or address range. The breakpoint is triggered if the application reads from any part of the specified memory range. Where supported by your debug target, you can also add data tests.

**HW Write**
Sets or modifies a write breakpoint at the specified memory location or address range. The breakpoint is triggered if the application writes to any part of the specified memory range.

**HW Access**
Sets or modifies an access breakpoint at the specified memory location or address range. The breakpoint is triggered when a memory address is accessed. This type of breakpoint enables you to perform hardware tests or to compare data values.

**HW Data Value Read**
Sets or modifies a breakpoint that is triggered if a specified data value is read from any address, and then detected by the debug hardware on the target processor.

**HW Data Value Write**
Sets or modifies a breakpoint that is triggered if a specified data value is written to any address, and then detected by the debug hardware on the target processor.

**HW Data Value Access**
Sets or modifies a breakpoint that is triggered if a specified data value is accessed at any address, and then detected by the debug hardware on the target processor.

For each of these types, use the HW Support group to specify how the match is configured.

To set up the hardware breakpoint you must enter:

**Location:**
Specifies the memory location where the new breakpoint is set. Where supported by your target hardware, use the options from the right-arrow menu to qualify the location (see Using ranges and masks on page 4-31 for details).
Value Match:

Enter the data value that triggers the breakpoint.

If you use this with data breakpoints, this compares the data value that is read or written. When used with an instruction hardware breakpoint, this can test the value of an instruction. That is, it can be used to find an instruction in a given range.

Where supported by your target hardware, use the options from the right-arrow menu to qualify the value match (see Using ranges and masks on page 4-31 for details).

Note

Depending on the Break/Tracepoint Type you select, the Location or Value Match fields might be unavailable. In this case, the field is grayed out. Also, in this example, the Class field is read-only and the drop-down arrow is unavailable.

HW Support

Where your debug target supports breakpoint tests in hardware, they can be managed and edited using this group. If enabled, the display lists currently available tests, for example for an ARM-based target:

- **DataSize**
  
  Supports testing **MAS** signals from the core. This enables you to test the size of data data bus activity.

- **Mode**
  
  Supports testing **nTrans** signals from the core. This enables you to test the data not translate signal to differentiate access between a User mode and a privileged mode.

- **Extern**
  
  Supports hardware breakpoints that are dependent on some external condition.

The current test is shown in round brackets, for example Ignore or Any.

To change a selected test, highlight the test, for example \texttt{Match=Mode:(Ignore)}, and then click **Edit Value** to change how the test is defined, shown in Figure 4-20 on page 4-31.
Click OK to confirm the new test and close the list selection box. The HW Support display list shows the new test.

Highlight a test and click Reset to restore the default settings.

### 4.5.2 Using ranges and masks

Where supported by your target hardware, you can qualify the location and value match entries using options available from the right-arrow menu.

Click the right-arrow at the side of the Location data field to display the **Address Range and Mask** menu, shown in Figure 4-21. Use options from this menu to specify an expression range or mask a group of instructions.

Click the right-arrow at the side of the Value Match data field to display the **Value Range and Mask** menu, shown in Figure 4-21. Use options from this menu to test a range of values or mask a range of data values.
**Note**

These menu options are only available where supported by your debug target and if you have specified an appropriate Break/Tracepoint Type.

Choose from the available options to set up your breakpoint:

**Address/Value Range**

Enter the start address, or data value, for the breakpoint then click this option to specify a range, for example the address range 0x800FF..0x80A00. The separators . are automatically inserted for you.

**Address/Value Range by Length**

Enter the start address, or data value, for the breakpoint then click this option to specify a range by length, for example the address range 0x800FF.+0x1111. The separators .+ are automatically inserted for you.

**Address/Value Mask**

Enter the address, or data value, for the breakpoint then click this option to specify a mask. RealView Debugger inserts the mask for you, for example 0x800FF $MASK=0xFFFF. Change this mask as required.

The mask is a bitwise-AND mask applied to the specified address, or data value, for example given the location 0x0111 and a mask 0x1001 the result is 0x0001.

The breakpoint triggers when the address, or data value, matches the given value after masking.

**NOT Address/Value Compare**

Enter the address, or data value, for the breakpoint then click this option to specify a NOT operation, for example $NOT 0x800FF.

Similarly, use this option to specify a range of addresses, or data values, to ignore, for example $NOT $0x0500..+0x100.

**Autocomplete Range**

Enter a symbol and then click this option to compute the end-of-range address based on the symbol size. For example, if you enter a function then the autocompleted range is from the start to the end of the function. Similarly, enter a global variable to see the end-of-range address autocompleted as the variable storage address plus variable size.
### Note

Many combinations of range, or mask, options are permitted. However, mixing range and mask generates a warning message to say that this is not permitted.

#### 4.5.3 Setting hardware breakpoints on a DSP-based debug target

The options available from the Set Address/Data Break/Tracepoint dialog box change if, for example, you are using a DSP-based debug target.

If you are working with an Oak-based debug target, the HW Support group offers a single option, shown in Figure 4-23. The **HWPassCount** enables you to specify the number of times the test point is passed before the breakpoint triggers.

![Figure 4-23 HW Support group using a DSP](image)

To change the hardware pass count, click **Edit Value** to edit how the test is defined, shown in Figure 4-24.

![Figure 4-24 Changing the hardware pass count in a DSP](image)

Click **Set** to confirm the count and close the prompt box. The HW Support display list shows the new test.

Highlight the test and click **Reset** to restore the default settings.

Where supported by your target hardware, you can qualify the location and value match entries using options available from the right-arrow menu (see *Using ranges and masks* on page 4-31 for details).
4.6 Using complex breakpoints

Select Debug → Complex Breakpoints from the Code window main menu to display the Complex Breakpoints menu, shown in Figure 4-25.

Complex breakpoints use advanced hardware support on your target processor, or as implemented by your simulator software. The breakpoint types available depend on the:

- hardware support provided by your target processor
- target vehicle used to maintain the connection.

Menu options are enabled where they are supported by your debug target. Check your hardware characteristics (see Viewing your hardware characteristics on page 4-5), and your vendor-supplied documentation, to determine the level of support for complex breakpoints.

4.6.1 Setting complex breakpoints

The Complex Breakpoints menu offers:

**HW Break if in Range...**

Displays a dialog box where you can specify a hardware breakpoint, shown in Figure 4-26.

Use this to set, or modify, a breakpoint at the specified location. The breakpoint is triggered if the PC is within the given address range.
Select the And if Data Value matches check box if you also want to test a data value to trigger the breakpoint. Enter the data value to test in the data field, shown in Figure 4-26 on page 4-34.

**HW While in func/range, Break if X...**

Displays a dialog box where you can specify a complex breakpoint that uses two breakpoint units, shown in Figure 4-27.

**Figure 4-27 HW While in func/range, Break if X dialog box**

Specify the function, or the address range, to test for the first breakpoint unit. The breakpoint triggers if the PC falls within the specified range.

Choose the type of breakpoint that you want to set for the second breakpoint unit, for example HW Read. You can click on the drop-down arrow to display a menu of possible breakpoint types.

Select the And if Data Value matches check box if you also want to test a data value to trigger the breakpoint. Enter the data value to test in the data field, shown in Figure 4-27.

**Note**

Setting a breakpoint this way displays the breakpoint as two entries in the Break/Tracepoints pane.

**HW Break if X, then if Y...**

Displays a dialog box where you can specify a complex breakpoint that uses two breakpoint units, shown in Figure 4-28 on page 4-36.
Use this dialog box to set, or modify, a breakpoint based on two conditions being met, that is X and Y. The breakpoint is set at the specified memory location or address range depending on the values read.

Choose the type of breakpoint that you want to set for the first breakpoint unit, for example **HW Read**. You can click on the drop-down arrow to display a menu of possible breakpoint types.

Specify the address range to test for the first breakpoint unit (X). The breakpoint unit triggers if the PC falls within the specified range.

Specify the second breakpoint unit (Y) in the same way.

--- Note ---

Setting a breakpoint this way displays the breakpoint as two entries in the Break/Tracepoints pane.

--- HW Break on Data Value match... ---

Displays a dialog box where you can specify a hardware breakpoint, shown in Figure 4-29.

Specify the range of data values to test for the breakpoint, for example the low value, and the high value, or use a mask. The breakpoint triggers if the PC falls within the specified range.
4.7 Using the Break/Tracepoints pane

The Break/Tracepoints pane provides a central point to manage all breakpoint operations. It enables you to:

- view all breakpoints currently set and their status (enabled or disabled)
- specify the command used to create a breakpoint
- edit an existing breakpoint specification, for example you might want to change the location, update the qualifiers, or change debugging actions when the breakpoint triggers
- copy the attributes of an existing breakpoint to create a new breakpoint at another location
- create new breakpoints as an alternative to using the Debug menu
- access the Break/Tracepoints and Pane menus to manage your breakpoint operations.

The Break/Tracepoints pane also enables you to manage tracepoints during Trace and Analysis operations, and to work with thread breakpoints when running in RSD mode. For full details see RealView Debugger v1.7 Extensions User Guide.

This section describes:
- Displaying the Break/Tracepoints pane
- Viewing breakpoints in the Break/Tracepoints pane on page 4-38
- Using the Break/Tracepoints menu on page 4-40
- Working in the Break/Tracepoints pane on page 4-41
- Using the Pane menu on page 4-42.

4.7.1 Displaying the Break/Tracepoints pane

Select View → Pane Views → Break/Tracepoints Pane from the Code window main menu to display the Break/Tracepoints pane, shown in Figure 4-30 on page 4-38.
If any breakpoints are set already, these are displayed in the new pane. To expand the tree and see the breakpoint details either:

- Click on the plus sign associated with a particular breakpoint entry.
- Double-click anywhere along the breakpoint entry (not on the check box).

The Break/Tracepoints pane also displays any other breakpoints you set, for example tracepoints or thread breakpoints.

If no breakpoints are set then the Break/Tracepoints pane is empty. As you create and set new breakpoints (or tracepoints), the pane is automatically updated to display each new breakpoint.

### 4.7.2 Viewing breakpoints in the Break/Tracepoints pane

The Break/Tracepoints pane shows entries in a tree view giving the Type and Value of each breakpoint you set, shown in the example in Figure 4-31 on page 4-39.
Each breakpoint is identified by a:

- colored icon to show the breakpoint type (see Viewing breakpoints in your code view on page 4-10 for details)
- check box to show if the breakpoint is enabled or disabled. You can click on the check box to change the state of the chosen breakpoint or breakpoint unit.

In Figure 4-31, an area of memory on the debug target has been designated ROM and so a BREAKINSTRUCTION command tries to auto-switch to a BREAKEXECUTION command if the given location is found to be in this part of memory. When the hardware breakpoint limit is reached for this debug target no further breakpoints can be set on code in this area, and a warning message is displayed.

--- Note ---

For details on the BREAKINSTRUCTION and BREAKEXECUTION commands see RealView Debugger v1.7 Command Line Reference Guide.

In the example in Figure 4-32 on page 4-40, different types of hardware breakpoints have been set, and disabled, during the session.
4.7.3 Using the Break/Tracepoints menu

If you have at least one breakpoint set, right-click anywhere along the entry (not on the check box) to display the Break/Tracepoints context menu, shown in Figure 4-33.

The Break/Tracepoints menu includes options to edit and manage your breakpoints:

- **Clear**  
  Removes the breakpoint selected in the Break/Tracepoints pane.

- **Disable**  
  This option changes the status of a selected breakpoint.
  
  Select **Disable** to disable a breakpoint you have set. This changes to **Enable** so that you can enable a breakpoint that has been disabled.

  Select, or unselect, the check box in the Break/Tracepoints pane to change the status in the same way.
Reset PassCounters/Then-Enables
Resets specific qualifiers for a given breakpoint and then re-enables it ready for triggering as appropriate.

Details... Displays an information box giving details of the selected breakpoint.

Break/Tracepoint Favorites...
Displays the Favorites Chooser/Editor where you can edit or delete entries, or select from your favorites to set a breakpoint.

Show Code The cursor moves to the location of the breakpoint in the appropriate code view:
- an open blue pointer marks the location in source-level view
- a filled blue pointer marks the location in disassembly-level view if there is no source.

4.7.4 Working in the Break/Tracepoints pane
If you have at least one breakpoint set, you can use the Break/Tracepoints pane to:

- edit a breakpoint:
  1. Right-click on a breakpoint in the list to display the Break/Tracepoints menu, shown in Figure 4-33 on page 4-40.
  2. Select Edit Break/Tracepoint... from the context menu to display the Set Address/Data Break/Tracepoint dialog box where you can edit the breakpoint or breakpoint unit.

- copy an existing breakpoint to create a new breakpoint:
  1. Right-click on a breakpoint in the list to display the Break/Tracepoints menu, shown in Figure 4-33 on page 4-40.
  2. Select Copy Break/Tracepoint... from the context menu to display the Set Address/Data Break/Tracepoint dialog box, populated with the relevant information about the chosen breakpoint.

      Edit the definition to create a new breakpoint, for example change the location of the breakpoint, or add a qualifier.

You can also use the Break/Tracepoints pane to edit or copy other breakpoints you set, for example tracepoints or thread breakpoints.
4.7.5 Using the Pane menu

Click on the drop-down arrow on the Break/Tracepoints pane title bar to display the Pane menu. This menu includes options that are also available from the Debug menu. The relationship between the two menus is summarized in Table 4-1.

<table>
<thead>
<tr>
<th>Break/Tracepoints pane, Pane menu</th>
<th>Code window, Debug menu</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Break from Function/Label list</td>
<td>Simple Breakpoints → Set from Function/Label list...</td>
</tr>
<tr>
<td>Address/Data...</td>
<td>Simple Breakpoints → Address/Data...</td>
</tr>
<tr>
<td>Set BreakIf...</td>
<td>Simple Breakpoints...</td>
</tr>
<tr>
<td>Named Break...</td>
<td>Simple Breakpoints → Named...</td>
</tr>
<tr>
<td>Processor Events...</td>
<td>Simple Breakpoints → Processor Events...</td>
</tr>
<tr>
<td>Break/Tracepoint History...</td>
<td>Simple Breakpoints → Break/Tracepoint History...</td>
</tr>
<tr>
<td>Break/Tracepoint Favorites...</td>
<td>Simple Breakpoints → Break/Tracepoint Favorites...</td>
</tr>
<tr>
<td>Show Break Capabilities of HW...</td>
<td>Complex Breakpoints → Show Break Capabilities of HW...</td>
</tr>
<tr>
<td>Clear All Break/Tracepoints</td>
<td>Simple Breakpoints → Clear All Break/Tracepoints</td>
</tr>
</tbody>
</table>

If you select the option Set BreakIf... from the Pane menu, a breakpoint type selection box is displayed containing all the breakpoints supported by your debug target.

---

**Note**

You can access some of the options from the Pane menu by right-clicking on a blank area inside the Break/Tracepoints pane. This enables you to create new breakpoints, select from your personal favorites or history list, or to see your hardware support.
4.8 Disabling and clearing breakpoints

You can temporarily disable breakpoints. This does not delete the breakpoint but means that you can enable it quickly for re-use in your current debugging session.

If you disable a breakpoint, you can still view it in the Src or Dsm tab where it is shown as a white disc. Any accompanying downward pointing arrows are colored light gray.

When you clear a breakpoint it is removed from the breakpoint list. To remove breakpoints from your favorites list, use the Favorites Chooser/Editor dialog box, see Saving breakpoints as favorites on page 4-45 for details of how to do this.

This section describes:
- Disabling breakpoints
- Clearing breakpoints on page 4-44
- Clearing all breakpoints on page 4-44.

4.8.1 Disabling breakpoints

You can enable and disable breakpoints from the Code window:

- In the Src tab:
  1. Right-click in the left margin, or on the line number, of a line marked with a red disc, or on the disc itself.
  2. Select Disable Break from the context menu.

- In the Break/Tracepoints pane, select the check box to unselect it so disabling the breakpoint.

- Right-click on the breakpoint in the Break/Tracepoints pane to display the Break/Tracepoints menu. Select Disable from the menu.

- Position the cursor at the breakpoint that you want to disable. Select Debug → Simple Breakpoints → Enable/Disable Break at Cursor from the main menu.

You can use these methods to:
- enable a disabled breakpoint, marked by a white disc
- enable, or disable, a conditional breakpoint, marked by a yellow disc
- enable, or disable, tracepoints and thread breakpoints.
4.8.2 Clearing breakpoints

You can clear breakpoints from the Code window in different ways:

- In the Src tab, double-click on the line number of a line marked with a red disc or on the disc itself.
- Right-click on the breakpoint in the Break/Tracepoints pane to display the Break/Tracepoints menu. Select Clear from the menu.
- Position the cursor at the breakpoint that you want to disable. Select Debug → Simple Breakpoints → Toggle Break at Cursor from the main menu.

You can use these methods to clear any type of breakpoint, for example a conditional breakpoint (marked by a yellow disc), or a tracepoint, or a thread breakpoint.

Note

If supported by your debug target, a RESET command can be used to carry out a processor reset. If you are using RVISS, a RESET command does not clear breakpoints or tracepoints.

4.8.3 Clearing all breakpoints

You can clear all the breakpoints set on a selected debug target in one operation from the Code window, either:

- Select Debug → Simple Breakpoints → Clear All Break/Tracepoints from the main menu.
- Click on the Pane menu in the Break/Tracepoints pane and select Clear All Break/Tracepoints.
4.9 Saving breakpoints as favorites

When you first start to use RealView Debugger on Windows, your personal favorites list is empty. You can create breakpoints and add them directly to this list or you can add breakpoints that you have been using in the current debugging session. This section explains the steps to do both:

- Creating a breakpoint favorite
- Saving existing breakpoints as favorites on page 4-46.

Note

You must connect to a target to enable RealView Debugger favorites.

4.9.1 Creating a breakpoint favorite

RealView Debugger keeps a record of all breakpoints that you set during your debugging session as part of your history file. By default, at the end of your debugging session, these processor-specific lists are saved in the file exphist.sav in your RealView Debugger home directory. This file also keeps a record of your favorites, for example Break Qualifiers, Break Actions, and Break/Tracepoints.

To create a new breakpoint and add it to your favorites list:

1. Select Debug → Simple Breakpoints → Break/Tracepoint Favorites... to display the Favorites Chooser/Editor dialog box.
2. Click New to display the New/Edit Favorite dialog box.
3. Enter the CLI command to create the breakpoint and, optionally, a short text description to identify the breakpoint for future use.
4. Click OK to confirm the entries and close the New/Edit Favorite dialog box. The Favorites Chooser/Editor dialog box is displayed with the newly-created breakpoint shown in the display list.
5. Use the available controls to update the new breakpoint favorite:
   - New Displays the New/Edit Favorite dialog box where you can create a second breakpoint.
   - Edit Highlight a breakpoint in the display list and click this button to display the New/Edit Favorite dialog box already populated with the breakpoint details ready to change.
   - Delete Highlight a breakpoint in the display list and click this button to delete the chosen breakpoint from your favorites list.
Add to List
Adds an existing breakpoint to your favorites list. See Saving existing breakpoints as favorites for details on using this button.

Set
Sets the specified breakpoint on your current debug target. An error message is displayed if a breakpoint already exists at this location.

Close
Closes the Favorites Chooser/Editor dialog box without setting a breakpoint on the current debug target. In this way you can build up a list of breakpoint favorites ready for your next debugging session.

Help
Displays the online help for this dialog box.

4.9.2 Saving existing breakpoints as favorites

If you have already set some breakpoints, RealView Debugger lets you choose which to add to your favorites list so that they are available to re-use in future debugging sessions or with other build target configurations of your application program.

To add existing breakpoints to your favorites list:

1. Select View → Pane Views → Break/Tracepoints Pane to display the Break/Tracepoints pane.
2. Highlight a breakpoint in the display list that you want to add to your favorites list.
3. Right-click and select Break/Tracepoint Favorites... from the Break/Tracepoints menu. This displays the Favorites Chooser/Editor.
4. Click Add to List to add the specified breakpoint to your favorites list. This displays the New/Edit Favorite dialog box.
   The Expression field contains the breakpoint details. The Description field enables you to enter a short text description to help you to identify the breakpoint. This text is optional.
5. Click OK to confirm the breakpoint details and close the dialog box.

The Favorites Chooser/Editor dialog box is updated to show the new breakpoint in the display list. Because this breakpoint is already set, click Close to close the dialog box. If required, you can set another breakpoint from your favorites list before closing the dialog box.

The edited favorites list is saved to your exphist.sav file, in your home directory, when you close RealView Debugger. For full details about the history file, see the chapter describing how to end your session in RealView Debugger v1.7 Essentials Guide.
Note

If you are using RealView Debugger on non-Windows platforms, the history file is only created if you create and save a favorite, for example a breakpoint or watchpoint. See Appendix B RealView Debugger on Sun Solaris and Red Hat Linux for details.
Working with Breakpoints
Chapter 5
Memory Mapping

This chapter describes how to manage target memory in the RealView® Debugger Process Control pane. It contains the following sections:

- About memory mapping on page 5-2
- Enabling and disabling memory mapping on page 5-4
- Setting up a memory map on page 5-5
- Viewing the memory map on page 5-7
- Editing map entries on page 5-11
- Setting top of memory and stack values on page 5-12
- Generating linker command files for non-ARM targets on page 5-13.
5.1 About memory mapping

Memory mapping is disabled by default when you first connect to your debug target, that is all memory is treated as RAM. If you are working with a suitable target you can enable memory mapping and then configure the memory using the Map tab in the Process Control pane.

Memory mapping might be useful depending on the debug target you are using and the applications you are developing:

- If you are working with a simulator, or evaluation board, to develop your application program, mapping memory ensures that you are not trying to load your image into memory that does not exist on the real hardware.

- If you are working with different types of memory, memory mapping enables you to load an image and specify the exact location of different sections of memory.

- If your application contains pointers or stacks, or other uses outside declared areas, it might not work correctly on the final hardware. Mapping memory as Auto makes these errors visible.

- You can declare and program Flash memory.

- Memory mapping can be used to generate linker information when developing a scatterloaded application. This is not supported by ARM® architecture-based targets.

- Memory mapping enables RealView Debugger to handle shared memory so that references are updated when modified by a different processor. This also prevents software breakpoints in shared code memory.

- If you are using a simulator that supports it, you can map memory to add wait states to obtain better cycle accuracy when profiling or measuring performance.

- Like register modeling, memory mapping provides a means for system developers to tell users about the memory configuration on boards, chips or in simulator models.

When working with memory mapping, you must be aware of the following:

- The top of memory value must be higher than the sum of the program base address and program size. If set incorrectly, the program might crash because of stack corruption or because the program overwrites its own code.

- There is no requirement that the top of memory address is at the true top of memory. A C or assembler program can use memory at higher addresses.
If you are working with a scatterloaded application, you must define the location of stack and heap in your code.

The default memory model for RealView ARMulator® ISS (RVISS) creates memory pages as required through the whole address space, and, therefore, the RVISS configuration file can specify a value for top of stack that is in high memory.

When an image is loaded to a debug target, the memory map is checked to confirm that it is valid to load to the locations specified in the executable program. Memory is loaded and then read back to verify a successful load and to confirm that genuine memory is present. Memory sections defined as auto are also updated to reflect the access type specified in the executable image.

By default, RealView Debugger tries to set a software breakpoint. However, where enabled, the memory map can be used to define the type of breakpoint.

The memory map is used to define how memory contents are color coded when displayed in the Memory pane.
5.2 Enabling and disabling memory mapping

To enable memory mapping before you load an image:

1. Connect to a suitable target.

2. Select View → Pane Views → Memory Map to display the Process Control pane and bring the Map tab to the front.

3. Right-click anywhere in the Map tab to display the context menu.

4. Click on the option Enable Memory Mapping.
   
   This enables memory mapping ready to load your image.

5. To disable memory mapping, right-click anywhere in the Map tab to display the Map context menu, shown in Figure 5-1.

The context menu contains different options depending on the state of your debug target. For example, if you load an image the menu contains the option Update Map based on Image.

6. Click on the option Enable Memory Mapping so that it is unselected.
   
   This disables memory mapping ready to load your image.

See Using the Map tab context menu on page 5-9 for details on the other options available from this menu.
5.3 Setting up a memory map

Mapping memory, before you load an image for debugging, enables you to have full access to all the memory on your debug target. You can do this:

- as part of your target configuration settings
- using an include file
- interactively using the Map tab
- by submitting the appropriate CLI commands.

In this example, you are going to set up memory manually for the current session. Target memory settings defined in this way are only temporary and are lost when you close down RealView Debugger. See the chapter describing configuring custom targets in *RealView Debugger v1.7 Target Configuration Guide* for details of how to configure a memory map as part of your target configuration settings.

With memory mapping enabled, set up your map in the Process Control pane:

1. Right-click on the Start entry in the Map tab to display the context menu, shown in Figure 5-1 on page 5-4.

2. Select **Add or Copy Map Entry...** to display the Add/Copy/Edit Memory Map dialog box, shown in Figure 5-2.

![Figure 5-2 Add/Copy/Edit Memory Map dialog box](image-url)
In RealView Debugger, memory mapping is defined by a start address and a block size by default, not by an end address. If you want to specify the end address, you must unselect the **End is inclusive Length (vs Addr.)** check box.

3. Edit the dialog box to change the default memory mapping as follows:
   a. Enter 0x0 in the Start Addr field.
   b. Enter 0x8000 in the End field to specify the block size.

4. Enter **Area before image** in the Description field to describe this block.

5. Click **OK** to confirm your changes and the Process Control **Map** tab is updated.

6. Set up the second block of memory using these settings:
   a. Start address = 0x8000
   b. Length = 0x8000
   c. Description = **Middle**

7. Click **OK** to confirm your changes and the Process Control **Map** tab is updated.

8. Set up the third block of memory using these settings:
   a. Start address = 0x10000
   b. Length = 0xFFFF0000
   c. Description = **Area after image**

9. By default, memory access is set to byte-size (8 bits) for ARM processors. Do not change this.

10. Click **OK** to confirm your changes and update the Process Control **Map** tab, shown in Figure 5-3.

![Memory mapped](image)
5.4 Viewing the memory map

The Process Control pane provides a view of the memory mapping for the debug target that is running your application. This section describes how to use the map:

- **Working with the Map tab**
- **Memory map configuration** on page 5-8
- **Using the Map tab context menu** on page 5-9.

5.4.1 Working with the Map tab

To view the memory map:

1. Connect to a suitable debug target.
2. Enable memory mapping (see *Enabling and disabling memory mapping* on page 5-4).
3. Load an image, for example `dhrystone.axf`
4. Select **View → Pane Views → Memory Map** to display the Process Control pane and bring the **Map** tab to the front.
5. Click on the plus signs to expand the entries, shown in Figure 5-4 on page 5-9.

The **Map** tab displays a tree-like structure for each component of the memory map showing the start address, size, and access rule. A one-line text description can also be included. The way that memory is shown depends on your debug target because RealView Debugger populates this tab from:

- built-in knowledge about the processor
- target configuration information
- the description in the target vehicle.

For example, if you are working with an ARM-based target, the first entry in the memory map shows `Start` (see Figure 5-4 on page 5-9) if the memory access rule is defined as *Any*. If you are using a DSP-based target, the first entry in the map shows the access rule for the type of memory at that location, for example `Prog`. Colored icons are used to show the type of memory defined, see *Display colors* on page 5-8.

With an image loaded, the **Map** tab is updated from details in the image itself. The memory map is also automatically updated if any registers change that affect memory mapping.
5.4.2 Memory map configuration

The memory map for a chosen processor is configured under the following headings:

**Type**
The type of memory page, for example Prog, I/O, Data. Where no such definition is given, the type is set to Any.

**Access**
Defines the access rules for the memory:
- **RAM** Memory is readable and writable.
- **ROM** Memory is read-only.
- **WOM** Memory is write-only.
- **NOM** No memory.
- **Auto** Memory is defined by the application currently loaded. If there is no application loaded, this shows NOM.

**Prompt**
You are prompted to confirm that this type of memory is permitted for the loaded application. If there is no application loaded, this shows NOM.

**Flash**
Memory is readable and, if a Flash programming method file (*.fme) is present, writable.

**Start**
The memory area is defined by the start address and the size. This defines the start address of the memory area.

**Size**
Defines the size of the memory area.

**Access size**
Defines the size of memory accesses.

**Filled**
This shows if a range contains data loaded from an application program.

**Description**
A text description of the purpose of the memory supplied as part of the automatic mapping. You can also supply this information yourself (see Setting up a memory map on page 5-5).

To see details about a map entry, right-click on the chosen entry and select Properties from the context menu. This displays a text description of the type of memory defined at this location.

**Display colors**
When using the Map tab to view the memory map, RealView Debugger uses color to make the display easier to read and to highlight the different memory definitions, shown in the example in Figure 5-4 on page 5-9.
In this example, memory has been mapped, using a Board/Chip definition file, to declare Flash. See the chapter describing configuring custom targets in *RealView Debugger v1.7 Target Configuration Guide* for details of configuring your target this way.

Colored icons enable you to identify the memory access defined:
- white (open) specifies *Any*, where no memory type is defined
- blue indicates RAM
- yellow indicates ROM
- green indicates Flash memory known to RealView Debugger
- red cross indicates no accessible memory is defined.

### 5.4.3 Using the Map tab context menu

The Map tab context menu, shown in Figure 5-1 on page 5-4, enables you to add new mappings or to update existing ones. The options are:

**Add or Copy Map Entry...**

Displays the Add/Copy/Edit Memory Map dialog box, shown in Figure 5-2 on page 5-5, where you can create a new map entry based on an existing entry.
Edit Map Entry...
Displays the Add/Copy/Edit Memory Map dialog box, shown in Figure 5-2 on page 5-5, where you can edit a map entry.

Update Map based on Image
Updates the memory map based on details held in the loaded image.

Update Map based on Processor
Reads those registers that affect the memory maps. This is done automatically for built-in map registers but might be required if you are using external map registers, defined in the target configuration settings. Select this option to force RealView Debugger to read the registers and so update the memory map.

Save Map to Linker Command file...
Writes the current map state to a new or existing linker command file. This inserts or edits the MEMORY definitions in the linker command file, allowing for proper loading of an application based on actual memory settings. See Generating linker command files for non-ARM targets on page 5-13 for full details of how to generate this file for non-ARM targets.

Delete Map Entry
Deletes the map entry under the pointer. There is no undo.

Reset Map (Delete All)
Redefines the memory map to the initial state based on processor information, target configuration information, and processor registers. There is no undo.
5.5 Editing map entries

To edit memory map settings using the Map tab:

1. Right-click on the first entry in the display list to display the context menu shown in Figure 5-1 on page 5-4.

2. Select the option Add or Copy Map Entry... to display the Add/Copy/Edit Memory Map dialog shown in Figure 5-2 on page 5-5.

3. Use the Start Addr field to define the starting location for the mapping. This already contains the start address for the chosen block, shown in the Map tab.

4. Use the End field to define the block size for the mapping. By default, this specifies the size of the memory block to be defined. If you want to specify the end address, rather than the block size, unselect the check box End is inclusive Length (vs. Addr) and then enter the address in the End field, for example 0xFFFFFFFF.

RealView Debugger automatically sets the size you specify. If the computed size does not fall on a page boundary an error dialog is displayed and you must resubmit the block size.

Entering a value of 0x0 remaps all memory from the starting address.

5. Highlight the access type in the display list, for example RAM.

6. Enter the memory type to be allocated, for example Any.

7. Enter a description of the new memory map settings, for example New test memory entry.

8. Click OK to confirm your new settings and to update the Map tab.

RealView Debugger displays a warning if you have entered any values incorrectly, for example a mismatch on start and end addresses. Correct these entries and click OK. When all entries are valid, the dialog box closes and RealView Debugger updates the Map tab.

5.5.1 Updating map entries based on registers

If you are connected to a debug target that uses register-controlled remapping, for example the ARM Integrator/AP board, the Map tab also displays the effects of any changes made to these registers. In this case, right-click on the first entry and select Update Map based on Processor from the context menu (shown in Figure 5-1 on page 5-4) to update the display based on these memory-mapped registers.
5.6 Setting top of memory and stack values

If defined, the top of memory variable specifies the highest address in memory that the C library can use for stack space. By default, a semihosting call returns stack base. Base of heap is then set to follow on directly from the end of the image data region.

You can create your own settings to specify the bottom of the stack address, the size of the stack, the bottom of the heap address, and the size of the heap. If you do not set these values manually, RealView Debugger uses default settings that are target-dependent. For ARM processors the default is 0x20000. If you are using RVISS to simulate an ARM target, the default setting for top of memory is 0x80000.

When you first connect to an ARM-based debug target, RealView Debugger displays a warning message in the **Cmd** tab:

*Warning: No stack/heap or top of memory defined - setting top_of_memory to 0x80000.*

To avoid this message, set permanent values for top of memory, stack base and limit, using the Connection Properties window. Configure your debug target and define these settings so that they are used whenever you connect with RealView Debugger. See the chapter describing configuring custom targets in *RealView Debugger v1.7 Target Configuration Guide* for details of how memory is configured in ARM-based debug targets, and for an example of how to set up your memory map.

You can set top of memory, and other ARM-specific runtime controls, as part of a project definition. However, the available options depend on your target processor and target vehicle.

You can also set top of memory, stack, and heap values on a temporary basis, that is for the current session, using the `@top_of_memory` register. To do this select **Debug → Memory/Register Operations → Set Register...** to display the Interactive Register Setting dialog box, where the register contents can be changed.

--- **Note** ---

If you are using the default RVISS to simulate an ARM processor, this is not a suitable target for setting top of memory in this way.
5.7 Generating linker command files for non-ARM targets

The memory map, shown in the Process Control Map tab, can be used to generate or modify a MEMORY section of a linker command file used when you build your program. This MEMORY directive information can then be used to position various sections of an application correctly. For details of how to set up such command options see the chapter describing customizing projects in RealView Debugger v1.7 Project Management User Guide.

--- Note ---

Linker command files are not currently supported by ARM targets.

---

To generate or modify a linker command file:

1. Right-click on the start address at the top of the entries and select Save Map to Linker Command File... from the context menu.

2. Specify the location of the file in the Select Linker Command File to Create or Modify dialog box. Remember that:
   - If the file already exists, RealView Debugger looks for a MEMORY directive block created previously and, if found, replaces that block.
   - If the file already exists, but no MEMORY directive block exists, RealView Debugger locates the first MEMORY section and inserts the MEMORY directive block before it.
   - If the file already exists, RealView Debugger makes a backup copy before updating the contents.
   - If there is no existing file, RealView Debugger creates the specified file ready to accept the MEMORY directive block.

The RealView Debugger linker command file generation process uses the built-in automatic memory mapping to generate data based on the connected target settings, for example the registers that control mapping.

The data recorded in the generated MEMORY block includes each internal RAM, ROM, and Flash section as appropriate. Each section is allocated a predefined name. All external memory added using the Map tab, or defined automatically from a loaded image, is allocated a name based on the characteristics of the memory.

The linker file format is processor-specific. If none is known, RealView Debugger uses a default format based on TI tools.

Example 5-1 on page 5-14 shows an example of a generated linker command file, in DSP format.
Example 5-1

`/* Linker Command file for the DSPxx processor */
/* This file was generated by RVDEBUG. You can edit everything
outside the MEMORY block defined by RVDEBUG. Updates by
RVDEBUG will only effect that block. */
/* RVDEBUG: generated data block. Updated Fri Apr 04 15:10:41 2003
Do not modify this block. Do not put MEMORY lines above
this line, put below end of this block. */
MEMORY
{
 /* Register @YYYY has (masked) value 0068 */
 PAGE 0: PDaRam: org=0x0080, len=0x177F /* internal 'Dual-Access' */
 PAGE 0: P_RAM: org=0x8000, len=0x032D /* external 'Sect .text' */
 PAGE 1: DMapReg: org=0x0000, len=0x005F /* internal 'Registers (mapped)' */
 PAGE 1: DScrDaRam: org=0x0060, len=0x001F /* internal 'Scratch Dual-Access' */
 PAGE 1: DLDaRam: org=0x0080, len=0x177F /* internal 'Dual-Access' */
 PAGE 1: D_RAM: org=0x8000, len=0x0085 /* external 'Sect .bss,.stack' */
 PAGE 1: DHRom(R): org=0xC000, len=0x3EFF /* internal 'Internal program-ROM' */
}
/* RVDEBUG: generated data above */

This example shows a combination of internal memory based on current register
settings (@YYYY) as well as external memory as defined by the loading of a program.

The following notes apply to this automatic file generation process:

• If RealView Debugger creates the linker command file a comments section is
  inserted in the file reporting that it was generated by RealView Debugger, shown
  in Example 5-1.

• If a file already exists and is being updated by RealView Debugger, the comments
  section is not inserted. RealView Debugger then inserts the generated commands
  above the original user-generated commands.

• If a file already exists and contains a RealView Debugger generated data block
  then this section is replaced when RealView Debugger updates the command file.

5-14 Copyright © 2002-2004 ARM Limited. All rights reserved. ARM DUI 0153E
Chapter 6
Working with Debug Views

This chapter describes how to monitor your program during execution using panes and views in the RealView® Debugger Code window. It contains the following sections:

- Working with registers on page 6-2
- Working with memory on page 6-15
- Working with the stack on page 6-26
- Using the call stack on page 6-32
- Working with watches on page 6-36.
6.1 Working with registers

The Register pane displays the contents of processor registers and enables you to change those contents. Where appropriate, the pane shows registers using enumerations to make it easier to read the details, and enables you to enter new values in this format. The Register pane updates the register values to correspond to the current program status each time the target processor stops.

The registers that are visible depend on your debug target, that is your processor and the debug interface. See your processor hardware documentation for details on processor-specific statistics. For information on different debug interfaces, see the appropriate documentation, for example:

- *RealView ICE User Guide*
- *Multi-ICE User Guide*
- *ARM MultiTrace User Guide*.

This section describes the options available when working with registers. It contains the following sections:

- *Displaying register contents*
- *Formatting options* on page 6-4
- *Changing register contents* on page 6-5
- *Viewing different registers* on page 6-6
- *Understanding the register view* on page 6-8
- *Viewing debugger internals* on page 6-10
- *Semihosting* on page 6-12
- *DCC semihosting* on page 6-13
- *Defining new registers* on page 6-13
- *Interactive operations* on page 6-14.

6.1.1 Displaying register contents

To examine the contents of registers:

1. Connect to your target and load an image, for example dhrystone.axf.

2. Select Edit → Editing Controls → Show Line Numbers to display line numbers. This is not necessary but might help you to follow the examples.

3. Select View → Pane Views → Registers to display the Register pane and bring the Core tab to the front.

4. Click on the Src tab to view the source file dhry_1.c.
5. Set a simple breakpoint by double-clicking on line 150.

6. Click **Go** to start execution.

7. Enter 5000 when asked for the number of runs.
   The program starts and then stops when execution reaches the breakpoint at line 150. The red box marks the location of the PC when execution stops.

8. The contents of the Register pane are updated to show the program status as the target stops, shown in Figure 6-1.

9. Click on the **Core** tab to view base registers for your target processor.
   The Register pane displays tabs appropriate to the target processor running your image and the target interface. Different target processors contain different registers and so the contents of this pane change depending on the target you are debugging.
   For more details on the contents of this pane see:
   - *Viewing different registers* on page 6-6 for examples of other tabs.
   - *Understanding the register view* on page 6-8 for information on registers.
   - *Viewing debugger internals* on page 6-10 for information on debugger internals and statistics.

10. Click **High-level Step Into** to execute one instruction and then stop. Register values that have changed, since the last update, are displayed in dark blue.
11. Click **High-level Step Into** a few more times and examine the register values as they change.

12. Right-click on a changed register and select **View Memory At Value** to use the chosen value as the starting address for a memory display.

   This displays the memory view in the last-used Memory pane. If a memory view is not visible, the default Memory pane, in the middle pane row, is used to display the view.

13. Monitor changes in the Register pane as you step through your program.

14. Double-click on the red marker disc to clear the breakpoint at line 150.

___ **Note** ____________

If you are using **RealView ARMulator® ISS** (RVISS), as in this example, registers in some ARM® cores are incompletely modeled, that is:

- ARM925T™, wait for interrupt
- ARM966E™-Sr2, TCM register size missing
- ARM946E™-Sr1, r13 trace PID
- ARM926EJ™-S, r13 context ID writing to the wrong register
- ARM720T™ r4, does not distinguish trace PID and FCSE r13.

___ **Display colors**

When using the Register pane to view register contents, RealView Debugger uses color to make the display easier to read and to highlight significant events:

- Black indicates values that are unchanged for the previous two updates.
- Dark blue shows those values that have changed since the last update.
- Light blue indicates a value that changed at the previous update.

___ **6.1.2 Formatting options**

You can change the way register values are displayed in the Register pane.

**For all registers**

Click the **Pane** menu to select formatting options for all registers currently displayed, for example, to display contents as values rather than enumerations or to display values in decimal format. This change applies to all tabs in the Register pane.

Use the option **Copy Pane** to take a snapshot of the top-level pane display. You can then copy this, for example into a text editor, so that you can compare registers and values.
For selected registers

You can change the display format for a single register while viewing other registers in the default format. Right-click on a chosen register, for example R3, to display the Register context menu.

This menu enables you to change the format of the chosen register, to view the properties, or to change the register contents, for example, select Increment to add one to the current value. You can also select from a list of previously used values to update the current contents.

The options offered on this menu depend on the register currently under the mouse pointer. Position your pointer over the Mode field of the CPSR register SVC and then right-click to display the Status register context menu.

You can use this menu to change the register contents, or pick from a list of values appropriate to this register.

Select Set Enumeration... to display the selection box shown in Figure 6-2.

![Figure 6-2 Setting enumeration values](image)

This dialog enables you to select a new value for the chosen register or to use an expression to define the contents. For example, highlight the entry <Set by Value/Expression...>. Click OK to display a box where you can enter the value or choose from a drop-down list.

6.1.3 Changing register contents

You can use in-place editing to change register contents in the Register pane. There are, however, other ways to set register values from the Code window:

- Set the contents of a register by pasting variables from other windows. For example, right-click on a value in the source-level view and select Copy from the context menu. Right-click in the register whose value you want to change and select Paste Value from the context menu.
Highlighted a value in the Src tab in the File Editor pane and then use drag-and-drop
to copy the expression into the contents of a chosen register in the Register pane.

- Select Debug → Memory/Register Operations → Set Register... from the Code
  window main menu to display the Interactive Register Setting dialog box.

Any register contents you change are displayed in blue.

Note
You can also set breakpoints on memory mapped registers but not on core registers,
because breakpoints are set on addresses and core registers do not have addresses.

Updating register contents
The Register pane updates automatically when register contents change. This is set by
default, as indicated by the checked option Automatic Update on the Pane menu.

If you change this default option, you must update the pane display manually. Select
Update Window Now from the Register pane context menu to update the view.
RealView Debugger updates the pane contents and displays changed values in blue for
improved readability.

6.1.4 Viewing different registers
The Register pane displays tabs and registers appropriate to your:

- target processor, that is different target processors contain different registers and
  so the contents of this pane change depending on the target you are debugging

- target interface, for example RealView ICE

- target configuration, for example if you are using a .bcd file, the pane includes
  extra tabs to provide extended target visibility.

In Figure 6-3 on page 6-7, the Register pane shows registers for an ARM940T core and
the ARM Integrator/AP board using Multi-ICE.
Figure 6-3 Registers for an ARM940T debug target

Here you can see the CP15 tab and other tabs relating to processor-specific operations, for example Data Regions, Cache Operations, and TLB Operations (Translation Lookaside Buffers). These are special registers and are described in full in the processor hardware documentation.

In the example in Figure 6-4, the Register pane displays register contents for an Oak-based debug target.

Figure 6-4 Registers for an Oak-based debug target

Here you can see the Status tab displaying the status registers.
Managing multiple targets

If you are licensed to use multiprocessor debugging mode you can access different registers on multiple debug targets at the same time. To do this, set up multiple Code windows, attach each window to a different debug target and then display the registers for each target. RealView Debugger enables you to set up several Register panes with different formatting options for each.

For details on setting up multiple Code windows and attaching to different debug targets, see the multiprocessing chapter in *RealView Debugger v1.7 Extensions User Guide*.

6.1.5 Understanding the register view

ARM processors support up to seven processor modes depending on the architecture version, for example USER, Supervisor, and FIQ. All modes, except User mode, are referred to as privileged modes.

ARM processors have thirty-seven registers arranged in partially overlapping banks. There is a different register bank for each processor mode, for example USER or FIQ. Using banked registers gives rapid context switching for dealing with processor exceptions and privileged operations.

RealView Debugger displays registers in named groups to reflect the ways registers are banked, for example USR, IRQ, and FIQ for an ARM7TDI® core. Click on the plus, or minus, sign to expand, or collapse, the view (shown in Figure 6-5 on page 6-9).
At first sight, it might appear that some registers are missing or that extra registers are visible for different processor modes. For example, FIQ contains R8, R9, R10, R11, R12, SP, LR, and SPSR. These are the registers in the bank for that processor mode that are banked out when an FIQ exception occurs.

However, USR also contains R8, R9, R10, R11, R12, SP, and LR. These are banked out registers indirectly accessible with LDM and STM instructions of the form:

\[
LDM \text{ rX, } (\text{r8-r14})^A
\]

\[
STM \text{ rX, } (\text{r8-r14})^A
\]

These examples use the ^ suffix to specify that data is transferred into or out of the USER mode registers. The register list must not contain the PC.

You must load banked out registers from memory to modify them, and you must store them to memory to read them. However, they might be of interest when writing task context switch code or a coprocessor emulator.

For full details on LDM and STM commands see the appropriate documentation, for example:

- RealView Compilation Tools Assembler Guide
- RealView Compilation Tools Developer Guide.
6.1.6 Viewing debugger internals

Depending on your debug target, debugger internals appear in tabs in the Register pane:

- Viewing statistics
- Viewing internal variables on page 6-11.

Viewing statistics

If you are using RVISS to simulate a target processor, and connect to RVISS using the RealView Connection Broker interface, the CycleCount tab displays a range of internal variables. What is shown depends on the processor that you have chosen to simulate, for example Figure 6-6 shows the display for an ARM7TDMI core.

![CycleCount Tab](image)

Figure 6-6 Viewing statistics (RVISS)

Figure 6-6 shows:

- **Instructions** The number of program instructions executed.
- **Core_Cycles** Internal core cycles indicating the time an instruction spends in the execute stage of the pipeline.
- **S_Cycles** The number of sequential cycles performed. The CPU requests transfer to, or from, the same address, or an address that is a word or halfword after the address used in the preceding cycle.
- **N_Cycles** The number of nonsequential cycles performed. The CPU requests transfer to, or from, the same address, or an address that is unrelated to the address used in the preceding cycle.
- **I_Cycles** The number of internal cycles performed. The CPU does not require a transfer because it is performing an internal function (or running from cache).
C_Cycles  The number of coprocessor cycles performed.

Total  The sum of the S_Cycles, N_Cycles, I_Cycles, and C_Cycles.

For information on what is shown for different target processors, see the appropriate documentation:

- RealView ARMulator ISS User Guide
- the hardware documentation for the target processor you are simulating.

Viewing internal variables

RealView Debugger uses internal variables just like any other program. These are stored with the image for persistence across different debugging sessions. The variables it uses depend on your debug target. If you are using RealView ICE, Multi-ICE, or the RDI interface of RVISS, you see a different set of debugger internals in the Debug tab. An example of the Debug tab for RealView ICE is shown in Figure 6-7.

![Figure 6-7 Viewing internal variables (RealView ICE)](image-url)
For information on what is shown for different target interfaces, see the appropriate documentation:

- *RealView ICE User Guide*
- *Multi-ICE User Guide*
- the hardware documentation for your target processor.

### 6.1.7 Semihosting

Semihosting enables code running on an ARM-based target to use facilities on a host computer that is running RealView Debugger. Examples of such facilities include the keyboard input, screen output, and disk I/O.

The EmbeddedICE® logic in ARM cores such as the ARM7TDMI, contains a *debug communications channel* (DCC), that enables data to be passed between the debugger and the target using the RealView ICE or Multi-ICE interface, without stopping the program or entering debug state.

DCC is a word-size communications mechanism implemented in the debug part of the ARM core as a number of registers in CP14. DCC takes the form of a 32-bit register through which data might be communicated between code running on the host and code running on the target.

--- **Note** ---

RealView Debugger does not currently support the use of channel viewers.

---

If you are using Multi-ICE to connect to a target, two modes of semihosting are supported:

- Standard semihosting, where the target processor enters debug state while the semihosting operation is performed.
- DCC semihosting, where a handler is automatically loaded to the target. Communication between the handler and the host is performed over the DCC.

DCC semihosting has two advantages over breakpoint-based semihosting:

- it is generally faster
- interrupts continue to be serviced because the target processor does not enter debug state.

Standard semihosting is the default choice because DCC semihosting is more intrusive on the target.
To specify semihosting set the variable `semihost_enabled`, shown in Figure 6-7 on page 6-11 and Figure 6-8.

--- Note ---

If you are using RealView ICE to connect to a target, semihosting does not work if RAM is not present at the exception vector.

### 6.1.8 DCC semihosting

Specify DCC semihosting by setting the variable `semihost_enabled` to `DCC`, shown in Figure 6-8. This means that the DCC semihosting software interrupt handler is installed in memory at the address specified by the `semihost_dcchndlr_addr` variable. It is essential that a region of memory starting at this address is available in target memory and is unused. The default address stored in this variable is 0x70000. You might have to change this to a lower value to suit the target memory.

![Figure 6-8 Specifying DCC semihosting (Multi-ICE)](image)

For full details on DCC semihosting with Multi-ICE, see *Multi-ICE User Guide*.

### 6.1.9 Defining new registers

RealView Debugger has built-in awareness of core registers and other standard registers for different processor families. These are displayed in the Register pane. However, you can define new ASIC registers using the `Advanced_Information` blocks in your target.
configuration settings. When configured, user-defined registers can be displayed in the Register pane in the same way as standard registers. See the chapter describing configuring custom targets in *RealView Debugger v1.7 Target Configuration Guide* for details.

### 6.1.10 Interactive operations

For full details on using interactive operations on register contents using the **Debug** menu, see Chapter 7 *Reading and Writing Memory, Registers, and Flash.*
6.2 Working with memory

The Memory pane displays the contents of memory and enables you to change those contents. On first opening, the pane is empty, because no starting address has been specified. If a starting address is entered, values are updated to correspond to the current program status each time your program stops.

This section describes the options available when working with memory displays and contains the following sections:

- Displaying memory contents
- Formatting options on page 6-16
- Operating on memory contents on page 6-20
- Data width on memory accesses on page 6-23
- Changing memory contents on page 6-23
- Managing multiple targets on page 6-25
- Interactive operations on page 6-25.

6.2.1 Displaying memory contents

To examine the contents of memory:

1. Select File → Reload Image to Target to reload the image dhrystone.axf.
   You can also reload an image using the Reload Image button on the Actions toolbar.

2. Click on the Src tab to view the source file dhry_1.c.

3. Set a simple breakpoint by double-clicking on line 150.

4. Click Go to start execution.

5. Enter 5000 when asked for the number of runs.
   The program starts and then stops when execution reaches the breakpoint at line 150. The red box marks the location of the PC when execution stops.

6. Select View → Pane Views → Memory to display the Memory pane.
   Start addresses can be set using in-place editing or using the context menu.

7. Right-click in the first address in the window to display the Address context menu.

8. Select Set New Start Address... to display the selection box shown in Figure 6-9 on page 6-16.
You specify the start address by giving an address in hexadecimal or by giving a C/C++ expression that RealView Debugger computes to obtain the starting address. You can use any valid expression using constants and symbols.

You can also use the drop-down arrow to select an expression from a browser or to re-use a value entered previously. The drop-down also gives access to your list of personal favorites where you can store a memory address for re-use in this, or future, debugging sessions.

In this example, memory addresses of interest are in the region of 000088A0 so set the start address to examine memory from this location.

Numbers entered here must start with a zero. This means that RealView Debugger can distinguish these entries from valid variable names.

9. Enter the required location, for example 0x00088A4, and then click Set to update the Memory pane.

The memory display is arranged in columns. The left-most column shows the memory address. The memory contents are shown in the other columns. The number of columns displayed varies depending on the size of the pane. Color coding is used to distinguish the type of memory being displayed, see Display colors on page 6-20 for details.

10. Monitor changes in the memory display as you step through your program.

11. Double-click on the red marker disc to clear the breakpoint at line 150.

### 6.2.2 Formatting options

You can change the way memory contents are displayed, and set the start address, using the Pane menu:

- **Copy**: If you have selected memory contents, use this option to copy the values to the clipboard ready to paste.
Update Window Now

If you have unselected the option Automatic Update, you can use this option to update the memory display manually. You can update the display using this option at any time when execution is stopped. This enables you to catch any memory updates made externally.

Recompute Expression Now

Where you have used a C/C++ expression to compute the start address, select this option to recompute the expression and, where necessary, start at the new location. Where you have used a fixed value to specify the start address, select this option to update only the pane contents.

Set New Start Address...

Select this option to enter a C/C++ expression to compute the start address. This displays the selection box shown in Figure 6-9 on page 6-16 with the current expression already displayed. Change the address and click Set to specify the start address for the memory view.

Previous Start Address

Uses a previous start address for displaying the contents of memory. The history list holds up to 16 previous start addresses added when:

- you enter a new start address or expression
- the current expression is recomputed to generate a new start address
- the start address is set from the Address context menu.

Set Number of Columns to show...

When the Memory pane first opens, the number of columns you can see depends on the size of the pane and is chosen so as to show an even number of bytes. Use this option to change the number of columns visible in the display. Use the selection box to show up to 32 columns in a single window. This number does not include the column used when the ASCII display option is selected, see Show ASCII below.

The default setting is 0 to configure the number of columns to fit the window size.

Automatic Update

Updates the memory display automatically, that is when:

- you change memory from anywhere in RealView Debugger
- program execution stops.

This is the default.
Recompute Expression on Update
Where you have used a C/C++ expression to compute the start address, select this option to recompute the expression when the pane contents are updated, see above, and start at the new location where necessary.

Timed Update when Running
If you are using an RTOS extension, the memory display can be updated at a specified time interval during program execution. Select this option to set this timer according to the update period specified below.
This is only available when you are in RSD mode and where supported by the underlying debug target (see the chapter describing RTOS support in *RealView Debugger v1.7 Extensions User Guide* for details).

Timed Update Period
Use this to choose the interval, in seconds, between window updates.
Any value you enter here is only used when the option Timed Update when Running is enabled.

Signed Decimal
Displays the memory contents as negative or positive values where the maximum absolute value is half the maximum unsigned decimal value.

Unsigned Decimal
Displays the memory contents from 0 up to the highest value that can be stored in the number of bits available.

Hexadecimal
This displays memory contents as hexadecimal numbers.

Hex, leading Zeros
Displays memory values in hexadecimal format including leading zeroes.
This is the default display format for data values in this pane.

Show ASCII
Adds another column to the Memory pane, on the right hand side, to show the ASCII value of the memory contents.
ASCII format displays column values as characters. The ASCII format is useful if, for example, you are examining the copying of strings and character arrays by transfer in and out of registers.
Any non-printable value is represented by a period (.).
Data formats

The Pane menu contains an extended panel to define how data values are displayed in the Memory pane. The display format used for viewing memory contents varies depending on the data types supported by your target processor:

**Minimum Access Size**

Displays memory contents in the format specified as the minimum memory access size for the target. This is the default.

**Bytes (8 bits)** Each column displays 8 bits of data.

**Half Words (16 bits)**

Each column displays 16 bits of data. Where your debug target is an ARM processor halfwords are aligned on 2-byte boundaries.

**Long Words (32 bits)**

Each column displays 32 bits of data. Where your debug target is an ARM processor long words are aligned on 4-byte boundaries.

**Long Long Words (64 bits)**

Each column displays 64 bits of data.

**Fixed (word size)**

Enables you to use fixed point format for displaying numeric values, that is based on the natural size for the debug target processor. The default format is unsigned and one less than the number of bits in the value.

**Fixed...** Displays a selection box that enables you to specify a fixed point format to display numeric values. The value entered here becomes the default display format for the pane.

**Floats (32 bits)**

Displays values in floating point IEEE format, occupying four bytes, for example:

\[ 2.5579302 \times 10^{-4} \]

**Doubles (64 bits)**

Displays values in floating point IEEE format, occupying eight bytes, for example:

\[ 4.71983561663 \times 10^{16} \]
Display colors

When using the Memory pane to view memory contents, RealView Debugger uses color to make the display easier to read and to highlight significant events:

- Black specifies RAM or memory that can be modified.
- Blue shows those contents that have changed since the last update. Light blue indicates a previous update.
- Yellow indicates the contents of ROM.
- Green indicates Flash memory known to RealView Debugger. Otherwise the values are displayed in yellow, indicating ROM.
- Red**** indicates one of:
  — no memory is defined at this location
  — memory at this location is defined as Auto meaning it is determined when loading your application program
  — memory is defined as prompt meaning that you are prompted to confirm the usage when loading the application.
- Red!!!! indicates that there has been a failure in performing the memory operation. Double-click, with the right mouse button, at this location to get an explanation of the problem.

6.2.3 Operating on memory contents

You can perform different operations on the memory displayed in the Memory pane using the context menus. The menu shown, and the options available, depend on the type of memory under the cursor when you right-click and on the valid licenses that you have. The color-coded display helps you to identify the memory type.

Note

If you right-click on a memory cell to access a context menu, any change you specify is made to the cell under the cursor. This is independent of any highlighted cells in the view.

Right-click on a memory address to display the Address context menu that provides options to set the start address:

Update (double right-click)

Updates the display in the Memory pane.
Set New Start Address...
Enables you to specify the starting address for the display of memory contents.

Previous
Enables you to use the previous starting address for the display of memory contents.

Recompute Expression
Where you have used a C expression to compute the start address, select this option to recompute the expression and, where necessary, start the display at the new location.

Right-click on a memory cell (or byte) shown as black or green, that is where the type is ROM, Flash, or modifiable, to display the menu shown in Figure 6-10.

![Figure 6-10 Memory value (RAM) context menu](image)

This menu contains the options:

**Update (double right-click)**
Updates the display in the Memory pane.

**Set Start Address from Content**
Enables you to use the cell contents as the starting address for the display of memory contents. This option is enabled when the cell contains a scalar the size of an address or pointer.

**Show Symbol from Content**
RealView Debugger looks up the address held in the cell and displays any symbol at that address.
Show Symbol at Address
Displays any symbol at the address contained in the cell, and not the contents.

Set to 0
Enables you to set the current memory cell to zero.

Increment
Enables you to add 1 to the contents of the memory cell.

Decrement
Enables you to subtract 1 from the contents of the memory cell.

Set Value...
Displays a prompt where you can enter a new value for the memory cell.
This new value is then entered and the memory display is updated.
A memory cell can also be changed using in-place editing.

Set Memory Interactive...
Enables you to use memory interactive operations available in RealView Debugger.

Fill Memory with Pattern...
Enables you to fill memory starting at this location.

Set Break At...
Displays the Set Address/Data Break/Tracepoint dialog box where you can specify a breakpoint on the current memory cell. The type of breakpoint offered depends on the type of memory at the chosen location. For example, if the memory is defined as ROM, RealView Debugger offers a hardware breakpoint first.

Set Trace Point...
Enables you to set tracepoints based on the current memory view.

Set Trace Range
Enables you to set tracepoints based on a range of values in the current memory view.

Right-click on a memory cell, or byte, that is yellow, that is where the type is ROM, to see the ROM-specific context menu that offers a subset of these options.
Memory errors

Where a memory cell contains !!!! (colored red), this shows that there has been an error in the memory operation. Right-click to display a menu with a single option:

Show Error Code...

Select this to display the error code returned from the debug target when the memory operation failed.

6.2.4 Data width on memory accesses

By default, RealView Debugger makes word-size accesses when reading or writing memory. You can change this in your memory map or by specifying the Memory_block settings in the Advanced_Information block in your target configuration file. For details see the chapter describing configuring custom targets in RealView Debugger v1.7 Target Configuration Guide.

6.2.5 Changing memory contents

To change memory contents:

1. Select File → Reload Image to Target to reload the image dhrystone.axf.
2. Click on the Src tab to view the source file dhry_1.c.
3. Set a simple breakpoint by double-clicking on line 150.
4. Click Go to start execution.
5. Enter 5000 when asked for the number of runs.

The program starts and then stops when execution reaches the breakpoint at line 150. The red box marks the location of the PC when execution stops.

6. Use the Pane menu, in the Memory pane, to set the start address to 0x8B00 and display the ASCII values, shown in Figure 6-11 on page 6-24.
The memory locations 00008B43-00008B46 contain the four hexadecimal values 0x32, 0x27, 0x4E, and 0x44 corresponding to the string ‘2’ND’.

Right-click in the value at 00008B43, that is 0x32, to display the context menu shown in Figure 6-10 on page 6-21.

This enables you to change the contents at the specified location. Select the option Set Value... from the context menu to display the selection box, shown in Figure 6-9 on page 6-16, where you can enter the new value.

Enter the required hexadecimal value 0x4E, or enter ‘N’, and click Set to update the memory display. You can use uppercase or lowercase to enter the new value.

The new value is displayed in blue and the ASCII value changes from 2 to N.

You can also use the drop-down arrow to select from a browser or to re-use a value entered previously. The drop-down also gives access to your list of personal favorites where you can store a data value for re-use in this, or future, debugging sessions.

7. Change the value at location 00008B44 from 0x27 to 0x6F (lowercase o).

8. Data values can be entered in a format that is different from the display format. Right-click in the location 00008B45 (0x4E), for example, enter the decimal value 32 and then click Set. The memory pane is updated with the new value, a space, displayed in the chosen display format.

9. You can also use in-place editing to change memory contents. Double-click in the value 0x44 (D) at location 00008B46 and change it to 0x32. Press Enter to confirm the new value.

If you press Escape then any changes you made in the highlighted field are ignored.

10. View the changed values in the memory display. Each new value is displayed in blue as the pane contents are updated.
11. View the changes you have made in the messages displayed when your program completes. The string “2’ND” has been replaced by “No 2”.

12. Double-click on the red marker disc to clear the breakpoint at line 150.

### 6.2.6 Managing multiple targets

If you are licensed to use multiprocessor debugging mode you can examine different memory views on multiple debug targets at the same time. To do this, set up multiple Code windows, attach each window to a different debug target and then display the memory contents for each target. RealView Debugger enables you to set up several Memory panes with different formatting options for each.

For details on setting up multiple Code windows and attaching to different debug targets, see the multiprocessing chapter in *RealView Debugger v1.7 Extensions User Guide*.

### 6.2.7 Interactive operations

You can also perform operations on memory contents including saving memory to a file, and reloading, and filling memory. See Chapter 7 *Reading and Writing Memory, Registers, and Flash* for details.
6.3 Working with the stack

The stack, or run-time stack, is an area of memory used to store function return information and local variables.

Executing a function sets up the stack. As the new function is called, a record is created on the stack including traceback details, and local variables. At this point these arguments and local variables become available to RealView Debugger and can be accessed through the Code window.

When the function returns, the area of the stack occupied by that function is recovered automatically and can then be used for the next function call.

In a typical memory-managed ARM processor, the memory model comprises:

- a large area of application memory starting at the lowest address (code and static data)
- an area of memory used to satisfy program requests, the heap, that grows upwards from the top of the application space
- a dynamic area of memory for the stack which grows downwards from the top of memory.

The Stack Pointer (SP) points to the bottom of the stack.

--- Note ---
Modifying a value in the stack might cause the application program to perform incorrectly or even to abort operation completely.

---

RealView Debugger can provide the calling sequence of any functions that are still in the execution path because their calling addresses are still on the stack. However, when the function is off the stack, it is lost to RealView Debugger. Similarly, if the stack contains a function for which there is no debug information, RealView Debugger might not be able to trace back past it.

This section describes ways of working with the stack:

- *Using the Stack pane* on page 6-27
- *Formatting options* on page 6-28
- *Operating on stack contents* on page 6-29
- *Context controls* on page 6-30
- *Setting a breakpoint* on page 6-31
- *Interactive operations* on page 6-31.
6.3.1 Using the Stack pane

The Stack pane enables you to monitor the contents of the stack as raw memory, and to make changes to those settings. This might be especially useful for assembly language programmers. The Stack pane shows the contents of the stack at the SP register which is always kept at the top-left of the display area. Use this pane to view changes as they happen in the stack.

The Stack pane enables you to follow the flow of your application through the hierarchical structure by displaying the current state of the stack. This shows you the path that leads from the main entry point to the currently executing function.

To view the Stack pane:

1. Select File → Reload Image to Target to reload the image dhrystone.axf.
2. Select View → Pane Views → Stack to view the Stack pane.
3. Click on the Src tab to view the source file dhry_1.c.
4. Set a simple breakpoint by double-clicking on line 150.
5. Click Go to start execution.
6. Enter 5000 when asked for the number of runs.
   The program starts and then stops when execution reaches the breakpoint at line 150. The red box marks the location of the PC when execution stops.
7. View the updated Stack pane, shown in Figure 6-12.

![Figure 6-12 Viewing the Stack](image)

The stack pointer, marked by SP, is located at the bottom of the stack. The frame pointer, marked by FP, shows the starting point for the storage of local variables.
8. Monitor changes in the Stack pane as you step through your program, for example by clicking **Hi-level Step Into**.

9. Double-click on the red marker disc to clear the breakpoint at line 150.

The stack is displayed in columns:

- **Address**: The left column contains the memory addresses of the stack. In some target processors that use a Harvard architecture, for example a DSP, a **D** is appended to show that this a data address. You must include this letter when specifying such an address as the starting address.

- **Value**: The right column displays the contents of the addresses in the stack.

As with the Memory pane, the memory display in the Stack pane is color-coded for easy viewing and to enable you to monitor changes.

### 6.3.2 Formatting options

You can change the way stack contents are displayed, and set the start address, using the **Pane** menu. This menu provides options to manage the display of stack contents during your debugging session, change the display format, and extract data from the pane for use in other panes or windows. Highlight an entry in the display and then choose from the list of available options:

- **Copy**: Copies the chosen entry from the list to the clipboard.

- **Previous Start Address**: Enables you to use the previous starting address for the display of memory contents.

- **Signed Decimal**: Displays the memory contents as negative or positive values where the maximum absolute value is half the maximum unsigned decimal value.

- **Unsigned Decimal**: Displays the memory contents from 0 up to the highest value that can be stored in the number of bits available.

- **Hexadecimal**: Displays memory contents as hexadecimal numbers.

- **Hex, leading Zeros**: Displays memory values in hexadecimal format including leading zeroes. This is the default display format for data values in this pane.
**Show ASCII** Adds another column to the Stack pane, on the right hand side, to show the ASCII value of the memory contents.

ASCII format displays column values as characters. The ASCII format is useful if, for example, you are examining the copying of strings and character arrays by transfer in and out of registers.

Any non-printable value is represented by a period (.)

**Data formats**

The **Pane** menu contains an extended panel to define how data values are displayed in the Stack pane. The display format used for viewing memory contents varies depending on the data types supported by your target processor:

**Bytes (8 bits)** Each column displays 8 bits of data.

**Half Words (16 bits)** Each column displays 16 bits of data. Where your debug target is an ARM processor halfwords are aligned on 2-byte boundaries.

**Long Words (32 bits)** Each column displays 32 bits of data. Where your debug target is an ARM processor long words are aligned on 4-byte boundaries.

**Long Long Words (64 bits)** Each column displays 64 bits of data.

### 6.3.3 Operating on stack contents

You can perform operations on the memory displayed in the Stack pane using the memory contents context menus, as described in *Operating on memory contents* on page 6-20.

**Changing the stack pointer**

As you step through your code, the default stack pointer is used, shown in Figure 6-12 on page 6-27. You can specify an expression or a register to use as the stack pointer from the Stack pane:

1. Right-click on an entry to display the context menu shown in Figure 6-13 on page 6-30.
Working with Debug Views

Figure 6-13 Setting a new stack pointer

2. Select Set New Start Address... to display the address prompt box.
3. Enter an expression or a register, for example @R9, as the new stack pointer.
4. Click Set to confirm your choice and close the address prompt box.

The new stack pointer is marked by Expression Pointer (EP), located at the bottom of the stack.

If you enter a blank expression, or remove the existing expression, in the address prompt box, RealView Debugger reverts to using the default stack pointer register. In this example, this was R13 shown in Figure 6-1 on page 6-3.

6.3.4 Context controls

There are two Context controls available from the Code window main menu:

Stack up

Moves up one stack level from the current scope location giving access to all local variables at that location. A stack level is determined by each calling function.

Stack down

Moves down one stack level from the current scope location giving access to all local variables at that location. A stack level is determined by each calling function.

You must use the Stack up control first, because the context is as far down the stack as possible.
6.3.5 Setting a breakpoint

To set a breakpoint in the Stack pane:

1. Right-click on the required value.
2. Select Set Break At... from the Stack Value context menu.
3. Complete the entries in the Set Address/Data Break/Tracepoint dialog box.
4. Click OK to close the dialog box and set the breakpoint.

RealView Debugger sets a breakpoint on a symbol address where it exists on the stack. As soon as you exit the function, the address is no longer meaningful. Do not, therefore, use such a breakpoint where execution runs past the function return call.

Unlike the Watch pane, the Stack pane acts like a snapshot of a chosen address. It does not track each invocation of a function and so is not able to track the chosen symbol.

6.3.6 Interactive operations

You can also perform other operations on memory contents using the Stack pane. See Chapter 7 Reading and Writing Memory, Registers, and Flash for details.
6.4 Using the call stack

Processors maintain a call stack for each processor in your debug target. If you are debugging multithreaded applications, a thread stack is also maintained.

As a program function is called, it is added to the stack. Similarly, as a function completes execution and returns control normally, it is removed from the stack. The stack, therefore, contains details of all functions that have been called but have not yet completed execution.

RealView Debugger includes features enabling you to monitor variables and access traceback as your debugging session develops:

- Using the Stack pane
- Using the Call Stack pane.

6.4.1 Using the Stack pane

The Stack pane enables you to monitor activity on the stack during program execution by giving access to the stack as raw memory. See Working with the stack on page 6-26 for full details.

6.4.2 Using the Call Stack pane

Use the Call Stack pane to follow the flow of your application through the hierarchical structure by examining the current status of functions and variables. This enables you to see the path that leads from the main entry point to the currently executing function at the top of the stack. You can also use the Call Stack pane to set breakpoints.

The Call Stack pane shows the:

- name of the function
- line number in the source file from which the function was called
- parameters to the function.

To use traceback:

1. Select File → Reload Image to Target to reload the image dhrystone.axf.
2. Click on the Src tab to view the source file dhrystone.c.
3. Set a simple breakpoint by double-clicking on line 150.
4. Click Go to start execution.
5. Enter 5000 when asked for the number of runs.
The program starts and then stops when execution reaches the breakpoint at line 150. The red box marks the location of the PC when execution stops.

6. Select View → Pane Views → Call Stack to view the Call Stack pane, shown in Figure 6-14.
   When you open the Call Stack pane, the first entry is 0x0000AFFC showing the address of the link register when the application starts. This is the default format for all variables in this pane.

7. Continue to step through your program. If the current line is a multistatement line, the Call Stack pane shows the information in the form of line and column details, shown in Figure 6-14.

   ![Figure 6-14 Multistatement details in the Call Stack pane](image)

8. Monitor changes in the Call Stack pane as you step through your program.

9. Double-click on the red marker disc to clear the breakpoint at line 150.

**Tabs in the Call Stack pane**

The Call Stack pane contains the tabs:

- **Call Stack** Displays details of the functions currently on the stack, shown in Figure 6-14.
- **Locals** Displays a list of the variables that are local to the current function.
- **Statics** Displays a list of static variables local to the current module.
- **This** In C++ the this pointer locates the object for which the member function was called. It is C++ specific.

**Using the Pane menu**

Click on the Pane menu button to display the Pane menu that contains options to:

- manage variables
- change the display format
change how contents are updated
extract data from the Call Stack pane for use in other panes or windows.

Click on the Call Stack tab, highlight an entry in the functions list and display the Pane menu options:

**Copy**  
Copy the chosen entry, where enabled.

**Timed Update when Running**  
Not available in this release.

**Timed Update Period**  
Not available in this release.

**Update Window Now**  
Updates the contents of the Call Stack pane. Use this when **Automatic Update** is disabled.

**Automatic Update**  
Refreshes the Call Stack pane as soon as a watchpoint is triggered and execution stops. This is enabled by default.

**Show char* and char[] as strings**  
Displays local variables of type char* and char[] (or casted) as strings. This is enabled by default.

**Show integers in hex**  
Displays all integers in hexadecimal format. Disabling this option displays all integers in decimal. This is enabled by default.

**Properties**  
Displays a text description of the item under the cursor.

### 6.4.3 Using context menus

The Call Stack pane contains several context menus depending on which entry is selected and the type of variable under the cursor.

For example, right-click a function in the Call Stack tab to display the Function context menu:

**Scope-to**  
Scopes to the chosen function.

**Break at...**  
Sets a breakpoint at the address defined by the chosen function, if this is permitted.
6.4.4 Stack controls

When working with the Call Stack pane there are two Context controls available from the Actions toolbar:

- Stack up
- Stack down.

See Context controls on page 6-30 for full details.
6.5 Working with watches

Use watches to monitor variables or to evaluate expressions during your debugging session:

- Setting watches in source-level view
- Working with the Watch pane
- Managing watches on page 6-39
- Saving watches as favorites on page 6-42.

6.5.1 Setting watches in source-level view

To set a watch:

1. Select File → Reload Image to Target to reload the image dhrystone.axf.
2. Click on the Src tab to view the source file dhrystone.c.
3. Right-click on a variable name, for example Run_Index at line 146, and select Watch from the Source Variable Name menu.
4. Set a second watch, for example on the variable Enum_Loc at line 155.

If you are working in source-level view and the Watch pane is visible, you can set watches in other ways:

- use the option Enter New Expression... from the Pane menu in the Watch pane
- drag-and-drop the variable to watch
- select a variable to watch, copy it, and paste it into the Watch pane.

6.5.2 Working with the Watch pane

The Watch pane enables you to view expressions and their current values. You can use the Watch pane to create breakpoints, or to change existing watched values.

Displaying the pane

Select View → Pane Views → Watch to view the Watch pane, shown in Figure 6-15 on page 6-38.

The Watch pane contains a series of tabs. The Watch1 tab is selected by default. You can set expressions on different tabs to make it easier to manage what is being watched during your debugging session. Click on the required tab to select it.
Expressions are listed in the order they were created. You can drag the column headings
to display the full name or value if required. Where an entry contains subentries, for
example an array, a plus sign is appended to the name. Click on this to expand the
display.

**Using the Pane menu**

Highlight an entry in the watched variables list and click the Pane menu button to
display the Pane menu. This contains:

- **Cut**
  Select this option to copy the chosen value to the clipboard and then
delete it. Where the entry has subentries, for example an array, select this
option to delete all subentries.

- **Copy**
  Select this option to copy the chosen value from the list to the clipboard.
  From here it can be copied into another tab in the Watch pane, or into
  another pane or window.

- **Paste**
  Pastes the contents of the clipboard into the chosen entry. This might be
  an entry from another tab in the Watch pane, or text from the File Editor
  pane or another application.

- **Delete**
  If you are using in-place editing, select this option to delete the chosen
  value, or character, from the entry without copying it to the clipboard.
  You can also delete a value by highlighting it and pressing the Delete key.

**Timed Update when Running**

If you are using an RTOS extension, this enables a timed update of the
Watch pane when the target processor is running. Select this option to
refresh the expressions list automatically where it contains memory
based expressions. Other expression types are not updated.

This is only available when you are in RSD mode and where supported
by the underlying debug target (see the chapter describing RTOS support
in *RealView Debugger v1.7 Extensions User Guide* for details).

**Timed Update Period**

Use this to choose the interval, in seconds, between window updates.

Any value you enter here is only used when the option **Timed Update when Running** is enabled.

**Update Window Now**

Updates the contents of the Watch pane. Use this when **Timed Update when Running** and **Automatic Update** have been disabled. Select this
to update the current tab.
Enter New Expression...
Displays a prompt box where you can enter the expression to be watched. This displays the name and current value in the current tab. Click the required tab before selecting this option.

Automatic Update
Refreshes the Watch pane as soon as execution stops. This is enabled by default.

Recompute Expression with same Context
If you watch an expression, the result is evaluated based on the current context. Select this option to recompute expressions using the context when set.

Show char* and char[] as strings
Displays values as strings where appropriate. This is enabled by default.

Show integers in hex
Displays all integers in hexadecimal format. Disabling this option displays all integers in decimal. This is enabled by default.

Properties
Displays a text description of the item under the cursor.

Viewing watches
As you set watches, expressions are added to the Watch pane, shown in Figure 6-15.

The entries correspond to the watches you set, in the order that you set them. Each expression is shown giving the Name and Value. You can expand the column headings by dragging on the boundary marker to make the details easier to read.
To delete an expression, highlight it and press Delete. There is no undo for this operation but saving the watches in your personal favorites list enables you to reinstate any deleted entry.

6.5.3 Managing watches

With a watch set, you might want to change the expression, change the display format used, or edit it directly to control how it is monitored. Using the Watch pane enables you to carry out these operations and gives access to context menus where editing options are available.

Figure 6-16 shows an example Watch pane, with several expressions already set.

One watched value is an array, shown by the plus sign appended to the variable name. Click on the plus sign to expand the view and display the array elements.

--- Note ---

If the chosen array is very large, RealView Debugger warns before expanding the view.

---

You can access context menus, inside the Watch pane, to control watch options and edit watches directly, see:

- Using the Name menu on page 6-40
- Using the Value menu on page 6-41
- Using the pane context menu on page 6-41
- Editing watches on page 6-42.
Using the Name menu

If you have set up expressions, right-click on a chosen entry Name to display the Name menu. This context menu provides options that operate on selected entries only:

**Update** Updates the displayed value for the chosen expression. This is applied in combination with any update options you set from the Pane menu.

**Format...** Displays a selection box to specify the format for the watched expression. Highlight the required format from the list of available formats. You can also cast top level expressions, including casting to array, for example char* and char[12].

Click OK to confirm your choice. This closes the selection box and the new format is applied to the chosen expression.

**Break at...** If enabled, select this option to set a breakpoint at this location.

**BreakIf...** If enabled, select this option to set a conditional breakpoint.

**Add from Favorites...** Displays the Favorites Chooser/Editor dialog box where data values saved in your personal favorites can be added to the Watch pane. See Saving watches as favorites on page 6-42 for details.

**View Memory At Value**
Select this option to use the chosen value as the starting address for a memory display.

This displays the memory view in the last-used Memory pane if visible. If a suitable pane is not visible, the default pane in the Middle pane row is used.

**View Memory At Address**
Select this option to use the address of the chosen item as the starting address for a memory display.

This displays the memory view in the last-used Memory pane if visible. If a suitable pane is not visible, the default pane in the Middle pane row is used.

**Properties...** Displays a text description of the item under the cursor.
Using the Value menu

With a watch set, you can right-click on a chosen entry Value to display the Value menu. This context menu provides options that operate on selected watches only:

**Update**  Updates the displayed value for the chosen expression. This is applied in combination with any update options you set from the Pane menu.

**Format...**  Displays a selection box where you can highlight the required format for the expression from the list of available formats.

**Set to 0**  Sets the value of the chosen expression to zero, where allowed. An error message is displayed if you try to set a value to zero when not permitted.

**Increment**  Adds 1 to the value of the chosen expression. An error message is displayed if you try to increment a value when not permitted.

**Decrement**  Subtracts 1 from the value of the chosen expression. An error message is displayed if you try to decrement a value when not permitted.

**Set from Favorites...**

Displays the Favorites Chooser/Editor dialog box where data values saved in your personal favorites list can be inserted into the specified location.

**Recent expressions**

The rest of this menu contains a list of recently-used variables and data values. You can re-use entries from this list as required.

Using the pane context menu

If you right-click anywhere inside an empty entry in the Watch pane, a short context menu is displayed. This provides the options:

**Update All**  Updates the details for all the expressions currently displayed in the Watch pane.

**Add from Favorites...**

Displays the Favorites Chooser/Editor dialog box where expressions saved in your personal favorites list can be added to the Watch pane.
Editing watches

You can use in-place editing to change expressions in the Watch pane, and to add new ones:

1. Double-click in the name you want to change, or press Enter if the item is already selected. The name is enclosed in a box with the characters highlighted to show they are selected (pending deletion).
2. Enter the new name, or move the cursor to change the existing expression, or add a cast.
3. Press Enter to store the new name.

If you press Escape then any changes you made in the highlighted field are ignored.

6.5.4 Saving watches as favorites

When you first run RealView Debugger after a Windows installation, all favorites lists, stored in your `exphist.sav` file, are empty. You can create watches and then add them directly to this list or you can add watches that you have been using in the current debugging session. This section explains the steps to follow to do both.

Creating a watch favorite

To create a Watch favorite, right-click inside a blank entry of the Watch pane to display the Name menu, and then select Add from Favorites... This displays the Favorites Chooser/Editor dialog box. If this is the first time you have used watches in RealView Debugger, the display list is empty.

To create a new watch and add it to your favorites list:

1. Click New to display the New/Edit Favorite dialog box shown in Figure 6-17.

![Figure 6-17 New/Edit Favorite dialog box](image)

2. Enter the expression to be watched, for example `Ptr_Comp`.
3. Enter a short text description to help you to identify the watch for future use, for example `my watch favorite`.

   This is optional.
4. Click **OK** to confirm the entries and close the New/Edit Favorite dialog box. The Favorites Chooser/Editor dialog box is displayed with the newly-created watch shown in the display list (shown in Figure 6-18 on page 6-44).

Duplicate entries are not permitted in the favorites list.

The Favorites Chooser/Editor dialog box contains the controls:

- **New** Displays the New/Edit Favorite dialog box shown in Figure 6-17 on page 6-42 where you can create a second watch.
- **Edit** Highlight a watch in the display list and select this option to display the New/Edit Favorite dialog box already populated with the watch details ready for editing.
- **Delete** Highlight a watch in the display list and select this option to delete the chosen watch from your favorites list.
- **Add to List** Adds an existing watch to your favorites list. See *Saving existing watches as favorites* for details on using this button.
- **Set** Sets the specified watch on your current debug target.
- **Close** Closes the Favorites Chooser/Editor dialog box without setting a watch, or changing the displayed list.
- **Help** Displays the online help for this dialog box.

**Saving existing watches as favorites**

With several watches already set, RealView Debugger lets you choose which to add to your favorites list so that they are available for re-use in future debugging sessions or with other target configurations of your application program.

To add existing watches to your favorites list:

1. Highlight an expression in the Watch pane that you want to add to your favorites list.

2. Right-click on the **Name** and select **Add from Favorites...** from the **Name** menu. This displays the Favorites Chooser/Editor, shown in Figure 6-18 on page 6-44.
Figure 6-18 Existing watches in the Favorites Chooser/Editor

The display list shows any watches already saved in your favorites list. The data field now shows the chosen expression.

3. Click **Add to List** to add the specified expression to your favorites list. This displays the New/Edit Favorite dialog box shown in Figure 6-19.

Figure 6-19 Adding a new favorite

The Expression field contains the chosen watch and you can enter a short text description to help you identify the watch favorite. This is optional.

4. Click **OK** to confirm the watch details and close the dialog box.

The Favorites Chooser/Editor dialog box is displayed showing the new watch in the display list. Because this watch is already set, click **Close** to close the dialog box. If required, set another watch from your favorites list before closing the dialog box.

The edited favorites list is saved to your `exphist.sav` file when you close RealView Debugger.

**Saving data values as favorites**

With several watches already set, you can right-click on the Value for a specified expression and display the **Value** menu. This context menu includes the option **Set from Favorites...** to specify a data value to be set.
Select this option to display the Favorites Chooser/Editor dialog box where you can:

- save an existing data value as an entry in your favorites list so that it can be re-used later in this debugging session
- take a data value already saved and use it to set the starting value for the specified watch.

Note

If you are using RealView Debugger on non-Windows platforms, the history file is only created if you create and save a favorite, for example a breakpoint or watchpoint. See Appendix B RealView Debugger on Sun Solaris and Red Hat Linux for details.
Chapter 7
Reading and Writing Memory, Registers, and Flash

RealView® Debugger includes options that enable you to work with registers and memory interactively during your debugging session. This chapter describes these options. It contains the following sections:

- About interactive operations on page 7-2
- Using the Memory/Register Operations menu on page 7-3
- Accessing interactive operations in other ways on page 7-5
- Working with Flash on page 7-6
- Examples of interactive operations on page 7-13.
7.1 About interactive operations

Use interactive operations to:

- set memory and registers
- patch assembly code (where supported by your debug target)
- read a file to memory
- write memory to a file
- verify memory against a file
- fill memory with a pattern of your choosing
- control Flash memory.

You can access all these features directly from the Debug menu. Selected operations are also available when you are working in panes. These are described in:

- Using the Memory/Register Operations menu on page 7-3
- Accessing interactive operations in other ways on page 7-5
- Working with Flash on page 7-6.

The last section in this chapter contains examples using interactive operations see:

- Examples of interactive operations on page 7-13.
7.2 Using the Memory/Register Operations menu

Use the **Debug** menu to carry out read and write operations on memory and registers:

1. Connect to your target and load an image, for example dhrystone.axf.
2. Select **Debug** from the Code window main menu to display the **Debug** menu.
3. Select **Memory/Register Operations** to display the menu shown in Figure 7-1.

![Figure 7-1 Memory/Register Operations menu](image)

This menu contains:

**Set Memory...**
Displays the Interactive Memory Setting dialog box where you can walk through memory and make changes where required.

**Patch Assembly...**
Enables you to patch assembly code during your debugging session. You can enter instructions in assembler format for patching directly into memory. You can use labels, including making new ones, and symbols. This is only available where supported by the underlying debug target, for example a DSP. This option is disabled by default.

**Set Register...**
Displays the Interactive Register Setting dialog box where you can walk through registers and make changes where required.

**Upload/Download Memory file...**
Displays the Upload/Download file from/to Memory dialog box where you can locate a specific file, of a given type, and read the contents into an area of memory, or write a memory range into the file for re-use, or verify that a memory range matches the file contents.

**Fill Memory with Pattern...**
Displays the Fill Memory with Pattern dialog box where you can specify a pattern that is used to write to a given area of memory.
Flash Memory Control...
Displays the Flash Memory Control dialog box where you can erase and write Flash memory. The Flash memory must be opened before trying to use this dialog box.
7.3 Accessing interactive operations in other ways

There are other ways to access interactive operations depending on where you are working:

- From the Memory pane
- From the Stack pane.

7.3.1 From the Memory pane

If you are working in the Memory pane, you can access memory operations:

1. Select View → Pane Views → Memory to display the Memory pane.
2. Right-click on a memory cell, or byte, that is black or green, that is where the type is ROM, Flash, or modifiable, to display the context menu.
3. Select the required option, for example Set Memory Interactive... to display the Interactive Memory Setting dialog box.

7.3.2 From the Stack pane

If you are working in the Stack pane, you can access memory operations:

1. Select View → Pane Views → Stack to display the Stack pane.
2. Right-click on a memory cell, or byte, to display the context menu.
3. Select the required option, for example Set Memory Interactive... to display the Interactive Memory Setting dialog box.
7.4 Working with Flash

To use RealView Debugger to control Flash memory on your chosen debug target, you must:

- configure your debug target to describe the Flash memory chip
- have access to an appropriate Flash Method (FME) file.

Depending on your current target, this might mean that you must first define the memory map to specify the Flash memory.

This section describes how to work with Flash memory during your debugging session. It includes:

- Flash definition files
- Flash Method files on page 7-7
- Flash examples on page 7-7
- Flash programming on page 7-8
- Using the Flash Memory Control on page 7-11.

7.4.1 Flash definition files

Files to enable you to use supported Flash devices are included in the root installation and are located in install_directory\RVD\Core\..\flash\examples\... Files are collected in subdirectories based on the target Flash device:

**Board-specific files**

Assembler files start with b_**, for example

`\IntegratorAP\b_IntegratorAP.s` Files starting with board_**, contain the ASCII format information for an FME file, for example

`\IntegratorAP\board_intel_arm.ame`.

These files include Flash memory programming files.

**Flash-specific files**

These programming files start with f_**, for example

`\IntegratorAP\f_intel_arm.s`, and flash_**, for example

`\IntegratorAP\flash_intel.ame`.

These files contain the algorithm for defining the Flash device and are used to create the FME file for your project.

To see how these files are used:

1. Start up RealView Debugger without connecting to a target.
2. Select Project → Open Project... to open the example project ...lash\examples\IntegratorAP\IntegratorAP.prj.
3. Select Project → Project Properties... to display the Project Properties window.
4. Left-click on *ASSEMBLE=default in the List of Entries, the left pane. This group is expanded and the contents are displayed in the Settings Values pane, the right pane.
5. Right-click on *Sources and select Explore from the context menu. This shows the programming file used to create the FME file for the project.
6. Left-click on *BUILD in the List of Entries. This group is expanded and the contents are displayed in the Settings Values pane.
7. Right-click on *Pre_Post_Link and select Explore from the context menu. This shows the link commands used to include the Flash definition files for the project.
8. Select File → Close Window to close the Project Properties window without making any changes.
9. Select Tools → Build to create the FME file as defined by the project, that is flash_IntegratorAP.fme.

7.4.2 Flash Method files

FME files include code to:
• enable you to write to the Flash on your debug target
• perform read, write, and erase operations
• describe the way the Flash is configured on the bus.

Example files are included for all supported Flash devices as part of the root installation.

7.4.3 Flash examples

The root installation contains a directory of examples for supported Flash devices. This area contains RealView Debugger project files that create FME files from assembler sources. All the Flash examples are located in:

install_directory\RVD\Core\...lash\examples

Files are collected in subdirectories based on the target Flash device.
7.4.4 Flash programming

Before you can use RealView Debugger to control a Flash device on your target, you must:

- describe the Flash memory chip in the memory map
- ensure that you have a correctly configured FME file.

The following example describes how to use the ARM® Integrator™ FME file to program Flash memory on the Integrator/AP board. If you have another target board with a standard AMD, ATME, or Intel Flash device you must create a board-specific assembler file and link that file to create an FME file before you can program the Flash memory. If you are using another type of Flash memory, you must also create the Flash programming routines.

The board-specific assembler and Flash memory programming files are installed as part of the root installation, in:

`install_directory\RVD\Core\...\flash\examples`

See Flash definition files on page 7-6 for details.

This example describes how to use the predefined Integrator/AP Flash configuration to write an image to the Flash memory on the Integrator/AP board, connected using Multi-ICE®. The example is split into sections, which must be executed in this sequence:

1. Defining your target
2. Programming the image into Flash on page 7-9.

Note

If you program the Flash on an ARM Integrator board using this release of RealView Debugger, you bypass the ARM Firmware Suite (AFS) Flash library system information blocks. These blocks are used by the AFS Flash Library, and are stored at the end of each image written to Flash. If you rely on these blocks to keep track of what is in the Flash memory of your target, keep a record of the state and recreate it after working through the example.

---

Defining your target

To configure the Flash target:

1. Start up RealView Debugger without connecting to a target.
2. Select File → Connection → Connect to Target... to display the Connection Control window.

3. Right-click on the entry Multi-ICE and select Connection Properties... from the context menu.
   This displays the Connection Properties window where you can view configuration settings stored in your board file.

4. Click on the entry CONNECTION=Multi-ICE, in the left pane, to display the settings values in the right pane.

5. Right-click on the entry BoardChip_name and select AP from the context menu.
   This means that the Integrator/AP board file is used for this connection.

6. Select File → Save and Close to close the Connection Properties window.

7. Connect to the target using the Connection Control window.

8. Click on the Log tab in the Output pane to see that RealView Debugger is using the Integrator/AP board file.

9. Select View → Pane Views → Memory Map to display the Map tab in the Process Control window, where you can see the Flash memory on the Integrator/AP board.

In this example, memory has been mapped, using a Board/Chip definition file, to declare Flash. See the chapter describing configuring custom targets in RealView Debugger v1.7 Target Configuration Guide for details of configuring your target this way.

**Programming the image into Flash**

To program the image, you ask RealView Debugger to write to the Flash memory region that you have defined in the board file. The Integrator Flash starts at memory address 0x24000000, so to write an image to Flash:

1. Build an image compiled to run with code at 0x24000000 and that has data in RAM.
   This example uses the dhrystone program stored in:
   $install_directory\RVDS\Examples\...
   Open the project and rebuild using modified linker options:
   a. Select Project → Project Properties to open the Project Properties window.
b. Click on the entry `BUILD` in the left hand pane and double-click on the `Link_Advanced` entry.

c. Right-click on the `Ro_base` entry in the right-hand pane and select **Edit Value** from the context menu. Enter the value 0x24000000.

d. Right-click on the `Rw_base` entry in the right-hand pane and select **Edit Value** from the context menu. Enter the value 0x8000.

e. Select **Save and Close** in the Project Properties window.

f. Select **Tools → Build** to build the project.

2. Select **File → Load Image...** to load your image. This displays the Load File to Target dialog box where you can locate the required image.

3. Click **Open** in the Load File to Target dialog box. This displays the Flash Memory Control dialog box, shown in Figure 7-2.

![Figure 7-2 Flash Memory Control dialog box](image)

4. Click **Write** to program the image into Flash.

   **Note**

   Wait for the write to complete before continuing.

5. Click **Close** to close the Flash Memory Control dialog box.

6. Click on the **Cmd** tab in the Output pane to see the Flash operations.
7. Select View → Pane Views → Memory Map to display the Map tab where you can see the Flash memory on the Integrator/AP board.

7.4.5 Using the Flash Memory Control

The Flash Memory Control dialog box consists of a display list, a read-only data field, a Flash Log, and control buttons:

**Flash:** Click this button to get details about the Flash. The data field next to the Flash button describes the type of Flash being used.

**Open Flash Blocks:**

- **AllOn** Selects all entries in the Open Flash Blocks list as indicated by a check in the accompanying check box. This enables you to carry out operations on all the open blocks.
- **AllOff** Unselects all entries in the Open Flash Blocks list as indicated by no check in the accompanying check box.

**Write** Writes data to the specified blocks of Flash.

**Erase** Erases every specified block of Flash. This normally sets every byte to 0x00 or 0xFF depending on the type of Flash being used.

**Cancel** Abandons any changes made to the specified blocks of Flash.

**Cancel All** Abandons all changes to the Flash contents.

**Details** Displays an information box describing the type of Flash being used.

**Erase Block before Write**

Select this check box to erase the Flash block before performing the write operation.

**Verify Block after Write**

Select this check box to verify the Flash block, against the data source, after performing the write operation.

**Use Current values for Unspecified data in block**

Specifies that the original contents should be maintained unless modified by the current operation. If unselected, the erase values are used.
This option should only be used if you are updating part of the Flash block and you want to retain the current values in the rest of the block. This check box is unselected by default. If you select this option, RealView Debugger reads and then writes the entire block. This might take some time to complete.

**Flash Log:** Displays a log of operations carried out on the selected Flash blocks.

**Close** Closes the Flash Memory Control dialog box.

**Help** Displays online help for this dialog box.

See *Setting Flash memory* on page 7-21 for an example of interactive Flash memory operations.
7.5 Examples of interactive operations

This section contains examples showing how to perform interactive operations on memory and registers:

- Setting memory
- Setting registers on page 7-15
- Downloading memory to a file on page 7-16
- Comparing memory with file contents on page 7-18
- Filling memory with a pattern on page 7-19
- Setting Flash memory on page 7-21.

7.5.1 Setting memory

To set memory contents:

1. Connect to your target and load an image, for example dhrystone.axf.
2. Select Edit → Editing Controls → Show Line Numbers to display line numbers. This is not necessary but might help you to follow the examples.
3. Click on the Src tab to view the source file dhry_1.c.
4. Set a simple breakpoint by double-clicking on line 150.
5. Click Go to start execution.
6. Enter 5000 when asked for the number of runs.
   The program starts and then stops when execution reaches the breakpoint at line 150. The red box marks the location of the PC when execution stops.
7. Select View → Pane Views → Memory to display the Memory pane.
   Start addresses can be set using in-place editing or using the context menu.
8. Right-click in the first address in the window to display the context menu.
9. Select Set New Start Address... and enter 0x000088A0 as the new start address.
10. Highlight the first byte at this address. In this example, this is 0x00.
11. Right-click and select Set Memory Interactive... from the context menu to display the Interactive Memory Setting dialog box, shown in Figure 7-3 on page 7-14.
12. Enter the required memory settings:

- **Type:** Select the display format. See *Formatting options* on page 6-16 for details of the memory formats.

- **Addr:** The address where the memory setting starts. Depending on the method used to display this dialog box, this field is already populated, as in this example. The address must be entered in hexadecimal format, for example 0x000088A0.

- **Value:** This read-only data field shows the current value, in hexadecimal and decimal formats, at the specified memory location.

- **Enter New Value:**
  Enter the value to be set at the current location, for example 0x08 or 8 (decimal).
  If the Memory pane is configured to update automatically, clicking *Set* immediately updates the memory contents. This is the default setting in the *Pane* menu.
  If you press Enter with no value in the Value data field, RealView Debugger moves automatically to the next, or previous, location.

- **Next Addr**
  Moves the target address to the next location by adding 1 to the address displayed in the Addr data field. This depends on the size of the current type.
Prev Addr
Moves the target address to a new location by subtracting 1 from the address displayed in the Addr data field. This depends on the size of the current type.

Clear New
Automatically clears any value entered in the Enter New Value data field ready to accept another value. By default, this feature is enabled.

Auto Inc Addr
If selected, this radio button instructs RealView Debugger to increment the target address automatically ready to accept a new setting.

Auto Dec Addr
If selected, this radio button instructs RealView Debugger to decrement the target address automatically ready to accept a new setting.

Log:
Displays a log of the changes you have made. This log is shown when you next display the dialog box.

13. See the log updated to show your change.
14. Click Close to close the Interactive Memory Setting dialog box.
15. Double-click on the red marker disc to clear the breakpoint at line 150.

Changed values are displayed in the Memory pane in the usual way. That is, updated values are displayed in dark blue or light blue, depending on when they last changed.

7.5.2 Setting registers

To set register contents:

1. Select File → Reload Image to Target to reload the image dhrystone.axf.
2. Click on the Src tab to view the source file dhry_1.c.
3. Set a simple breakpoint by double-clicking on line 150.
4. Click Go to start execution.
5. Enter 5000 when asked for the number of runs.

The program starts and then stops when execution reaches the breakpoint at line 150. The red box marks the location of the PC when execution stops.
6. Select View → Pane Views → Registers to display the Register pane.
7. Select **Debug → Memory/Register Operations → Set Register**... from the Code window main menu to display the Interactive Register Setting dialog box.

This dialog contains almost the same controls as the Interactive Memory Setting dialog box described in *Setting memory* on page 7-13.

8. Set up the required register settings:

   **Register**: Enter the register to change, for example @R4. Press Enter to confirm your choice.

   If required, use the drop-down arrow to select a previously used register from the stored list.

   **Value**: This read-only data field shows the current value, in hexadecimal and decimal formats, for the specified register.

   **Enter New Value**: Enter the value to be set, in hex or decimal, for example 0xCC4.

   All changed registers are displayed in blue.

9. See the log updated to show your change.

10. Click **Close** to close the Interactive Register Setting dialog box.

11. Double-click on the red marker disc to clear the breakpoint at line 150.

### 7.5.3 Downloading memory to a file

To download a memory range into a file:

1. Select **File → Reload Image to Target** to reload the image dhrystone.axf.

2. Click on the **Src** tab to view the source file dhrystone.c.

3. Set a simple breakpoint by double-clicking on line 150.

4. Click **Go** to start execution.

5. Enter 5000 when asked for the number of runs.

   The program starts and then stops when execution reaches the breakpoint at line 150. The red box marks the location of the PC when execution stops.

6. Select **View → Pane Views → Memory** to display the Memory pane.

7. Select **Debug → Memory/Register Operations → Upload/Download Memory file**... from the Code window main menu to display the Upload/Download file from/to Memory dialog box, shown in Figure 7-4 on page 7-17.
8. Specify the operation and set up the controls, as follows:
   a. Select the **Save Memory into File** radio button. This instructs RealView Debugger to access the specified memory block, read the contents, and write them to the given file.
   b. In the **File** text box, enter the full pathname of the file to use to read/write memory values.
   c. In the **Type of File** section of the dialog, select the data type to be used in the specified file where:
      - **OBJ** specifies an object file in the standard executable target format, for example ARM-ELF for ARM-based targets
      - **raw** specifies a data file as a stream of 32-bit values
      - **rawhw** specifies a data file as a stream of 16-bit values
      - **rawb** specifies a data file as a stream of 8-bit values
      - **ascii** specifies a space-separated file of hexadecimal values.
   d. Define the start location of the memory block.
      When writing memory, specify a range as an address range or as a start address and length, for example:
      - 0x88A0 ..0x8980
      - 0x88A0 ..+0x14
      If required, use the drop-down arrow to select a previously used location from the stored list.

   **Note**
   If you are reading from a file to memory, you must specify a start location. The range can be left blank where the data type is not binary.
If you are writing to a file from memory, you must specify a start location and a range.

9. Click **Apply** to create and write the specified file.
10. Click **Close** to close the Upload/Download file to/from Memory dialog box.
11. Double-click on the red marker disc to clear the breakpoint at line 150.

**Note**

If you are writing memory to a file and the specified file already exists, RealView Debugger warns of this and asks for confirmation before overwriting the file contents.

RealView Debugger warns you if the memory transfer is going to take a long time to complete. When reading or writing memory contents, you must be aware that:

- There is no limit on the size of file that RealView Debugger can handle.
- The time taken to complete the operation depends on the access speeds of your debug target interface.

### 7.5.4 Comparing memory with file contents

To verify a file:

1. Ensure that you have downloaded a memory range into a file as described in *Downloading memory to a file* on page 7-16.

2. Select **Debug → Memory/Register Operations → Upload/Download Memory file...** from the Code window main menu to display the Upload/Download file from/to Memory dialog box (see Figure 7-4 on page 7-17).

3. Select **Verify Memory and File**.

4. Specify the file to be compared.

5. Specify the start address to be compared. This must be within the range specified in the memory file.

6. Click **Apply** to compare the file contents with the specified memory block.

7. Click on the **Cmd** tab in the Output pane and view the results, for example:

   ```
   verifyfile,ascii,gui "D:\ARM\RVD\Test_files\memory_file_3"
   Mismatch at Address 0x000088B6: 0x8E vs 0x8F
   ```
The first mismatch is identified and the location reported. Any mismatches after this location are not reported.

8. Click **Close** to close the Upload/Download file from/to Memory dialog box.

9. Double-click on the red marker disc to clear the breakpoint at line 150.

### 7.5.5 Filling memory with a pattern

To fill a specified area of memory with a predefined pattern:

1. Select **File** → **Reload Image to Target** to reload the image `dhrystone.axf`.
2. Click on the **Src** tab to view the source file `dhry_1.c`.
3. Set a simple breakpoint by double-clicking on line 150.
4. Click **Go** to start execution.
5. Enter 5000 when asked for the number of runs.
   The program starts and then stops when execution reaches the breakpoint at line 150. The red box marks the location of the PC when execution stops.
6. Select **View** → **Pane Views** → **Memory** to display the Memory pane.
7. Select **Debug** → **Memory/Register Operations** → **Fill Memory with Pattern...** from the Code window main menu to display the Fill Memory with Pattern dialog box, shown in Figure 7-5.

![Figure 7-5 Fill Memory with Pattern dialog box](image-url)

8. Set up the required memory settings:
   - **Start:** Enter the start address for the memory range to be filled, for example 0x88A0.
     If required, use the drop-down arrow to select a previously used start address from the stored list.
End/LEN: By default, the memory area that is filled is defined by a start address and a length. Enter the length of the memory block to be filled in this data field, for example 14 (decimal).

The specified length must be given relative to the data type given in the Size data field specified below.

If the Use Length (Count) check box is unselected, you can specify the address that marks the end of the memory block.

Size: Enter the data type to be used in the file where:

- natural indicates the format specified as native for the debug target
- byte indicates support for 8-bit signed and unsigned byte form
- half-word indicates support for 16-bit signed and unsigned halfwords
- long-word indicates support for 32-bit signed and unsigned words.

Use Length (Count)

By default, the memory block to be filled is defined by a start address and the length. If this check box is unselected you can specify an address to mark the end of the filled block.

Pattern: Enter the pattern to be used as the fill, for example:

"AB"

0,1,0,1,0

9. Click OK to confirm your settings and close the Fill Memory with Pattern dialog box. Memory contents are rewritten and the Memory pane is automatically updated with changed values displayed in blue.

10. Double-click on the red marker disc to clear the breakpoint at line 150.

When filling memory blocks, you must be aware of the following:

- All expressions in an expression string are padded or truncated to the size specified by the Size value if they do not fit the specified size evenly.

- If the number of values in an expression string is less than the number of bytes in the specified address range, RealView Debugger repeats the pattern and so might fill an area in excess of the specified block, for example specify a pattern of 10 bytes and a fill area of 16 bytes. RealView Debugger repeats the pattern twice and so fills a block of 20 bytes.

- If more values are given than can be contained in the specified address range, excess values are ignored.
If a pattern is not specified, RealView Debugger displays an error message.

7.5.6 Setting Flash memory

Flash memory blocks open for access when you write to Flash, for example when you load an image. This displays the Flash Memory Control dialog box (see Using the Flash Memory Control on page 7-11 for details).

To write to Flash memory interactively:

1. Connect to your target, load an image, and write to Flash, as described in Working with Flash on page 7-6.

2. Select View → Pane Views → Memory to display the Memory pane. Start addresses can be set using in-place editing or using the context menu.

3. Right-click in the first address in the window to display the context menu.

4. Select Set New Start Address... and enter 0x24000000 as the new start address. This is colored green, indicating Flash.

5. Right-click in the first byte at this address.

6. Select Set Value... from the context menu.

7. Enter the new value 0xA0 at the prompt.

8. Click Set to confirm this value. This displays the Flash Memory Control dialog box to enable you to access the open Flash, shown in Figure 7-2 on page 7-10.

9. If you want to view details of the Flash memory, click Flash in the Flash Memory Control dialog box. This displays details of the Flash memory, as shown in Figure 7-6 on page 7-22.
10. In the Flash Memory Control dialog box, ensure that the **Erase Block before Write** option is checked.

11. Click **Write** to write to the chosen Flash location. Monitor the changes in the Memory pane as memory is updated. The Flash Log confirms the Flash operation.

12. Click **Close** to close the Flash Memory Control dialog box.

--- **Note** ---

You can also select **Debug → Memory/Register Operations → Flash Memory Control...** from the Code window main menu, to display the Flash Memory Control dialog box during your debugging session.
Chapter 8
Working with Browsers

RealView® Debugger provides list browsers to help with debugging tasks and monitor your program during execution. This chapter describes how to access these lists from the Code window. It contains the following sections:

- Using browsers on page 8-2
- Browsing modules and files on page 8-4
- Browsing functions on page 8-7
- Browsing variables on page 8-11
- Specifying browser lists on page 8-14
- Browsing C++ classes on page 8-16
- Other routes to the browsers on page 8-18.
8.1 Using browsers

Browsers enable you to search through your source files to look for specific structures and to monitor their status during program execution.

Browsers are available for:
- project modules and files
- functions
- variables
- C++ classes.

RealView Debugger uses scope to determine the value of a symbol. Any symbol value available to a C or C++ program at the current PC is also available to RealView Debugger.

Variables can have values that are relevant within:
- a specific class only, that is class scope
- a specific function only, that is local scope
- a specific file only, that is static global scope
- the entire process, that is global scope.

For full details on scope and scoping rules see the chapter describing working with the CLI in RealView Debugger v1.7 Command Line Reference Guide.

8.1.1 Accessing list browsers

To access the browsers:

1. Connect to your target and load an image, for example dhrystone.axf.
2. Select the Find menu from the Code window main menu.

This menu displays the main searching facilities available from RealView Debugger. It also includes:
- Module/File List
- Function List
- Variable List.

Where supported by your debug target, you can also access a list of memory mapped registers when you use the Set Address/Data Break/Tracepoint dialog box to set a breakpoint, see Using simple breakpoints on page 4-13 for details.
In addition, RealView Debugger includes a Symbol browser to view C++ class objects that can be accessed through the Symbol Browser pane. Either:

- Select View → Pane Views → Symbol Browser from the default Code window main menu.
- Click on the Pane Content menu in a chosen pane, for example the Watch pane, and select Symbol Browser.

The rest of this chapter describes the RealView Debugger list browsers in more detail.
8.2 Browsing modules and files

Using the Module/File browser enables you to examine the different files and modules that go to make up your program and how these components are accessed during program execution. In this way you can locate errors during your debugging session.

To display the Module/File List, shown in Figure 8-1, select **Find → Module/File List...** from the Code window main menu.

![Figure 8-1 Module/File List dialog box](image)

The Module/File List dialog box displays, in order of appearance, all the modules and files in the current program. Each entry in the list shows the module name and then the filename, if known, for example:

@dhystone\DHRY_2 - dhry_2.c

The program name is attached at the start using @, for example @dhystone\. 

Module names qualify symbolic references. The module name is usually the filename without the extension. All module names are converted to uppercase by RealView Debugger. If the extension is not standard, the extension is preserved, and the dot is replaced with an underscore, for example sample_arm.c is converted to SAMPLE_ARM, and sample_arm.h is converted to SAMPLE_ARM_H.

If two modules have the same name then RealView Debugger appends an underscore followed by a number to the second module, for example SAMPLE_1. If there is a third module this becomes SAMPLE_2 and so on for any additional modules.

Following this convention avoids any confusion with the C dot operator indicating a structure reference.

This section describes:
- **Specifying the list** on page 8-5
- **Scoping to a module or file** on page 8-5
- **Closing the browser** on page 8-6.
8.2.1 Specifying the list

When you first open the Module/File List dialog box, the list entries are determined by the default search entry ISearch but you can decide which modules and files are displayed by applying a search filter.

See Specifying browser lists on page 8-14 for details of how to specify the list for the chosen browser.

8.2.2 Scoping to a module or file

To scope to a module:

1. Click on the Src tab to view the source file dhry_1.c.
2. Select Find → Module/File List... to display the Module/File List dialog box shown in Figure 8-1 on page 8-4.
3. Select a module from the list.
4. Click Scope to scope to that module.

The Src tab is updated to show that the scope is forced, as illustrated in Figure 8-2.

The location in the source file is identified by:
- placing a blue pointer to the left of the line number in the File Editor pane
- enclosing the line of code in a red box showing the location of the PC.

If the file is not already open in the File Editor pane, RealView Debugger opens it automatically, the window scrolls to the right place, and the scope is adjusted. The Module/File List dialog box closes.
8.2.3 Closing the browser

If you scope to a new module or file, the Module/File List dialog box closes automatically. Otherwise, click Cancel to exit the browser without adjusting the scope.
8.3 Browsing functions

Use the Function browser to examine the different functions that go to make up your program.

To display the Function List, shown in Figure 8-3, select Find → Function List... from the Code window main menu.

![Function List dialog box](image)

The Function List dialog box lists all the functions, ordered by module name, in the current program. Each entry in the list shows the filename, if known, and then the function name, for example:

```
DHRY_2\Func_3 of @dhrystone
```

This section describes:
- Specifying the list
- Refining the list on page 8-8
- Viewing details of a function on page 8-8
- Scoping to a function on page 8-9
- Setting a breakpoint on page 8-9
- Closing the browser on page 8-10.

8.3.1 Specifying the list

When you first open the Function List dialog box, the list entries are determined by the default search entry ISearch but you can decide which functions are displayed by applying a search filter.

See Specifying browser lists on page 8-14 for details of how to specify the list for the chosen browser.
8.3.2 Refining the list

The Function List dialog box contains check boxes that enable you to refine what is displayed in the list box:

- **Publics**: Displays global or public functions with scope over all parts of the program.
- **Statics**: Displays static functions.
- **Labels**: Displays code labels with scope over the entire function.

8.3.3 Viewing details of a function

Highlight a function in the display list. The Function List dialog box contains controls to display more details of this function or to perform specific debugging activities:

- **Disasm**: Displays the memory address in hexadecimal and assembly code in the Dsm tab starting at the specified memory location.
- **Source**: Displays the source code in the corresponding Src tab beginning at the specified line number or procedure name.
- **Break**: Sets a software breakpoint at the specified function, defined as a location in the image.
- **GoTo**: Sets a temporary breakpoint at the specified function. The program then executes from the current position of the PC. When execution reaches the breakpoint it stops. The temporary breakpoint exists only for the duration of this run and so is not shown in the Break/Tracepoints pane. Similarly, there is no red breakpoint marker shown in the source file. If the program stops before it reaches the temporary breakpoint, you must reinstate it before restarting the run.
- **Type**: Displays type information for the selected function. This information is displayed in a style similar to the source language.
- **Info**: Submits a CExpression command to calculate the value of a given expression by calling the specified target function. The function is converted into a debugger call macro, and strings and arrays passed to the target function are copied onto a stack maintained by RealView Debugger. A function called in this way behaves as though it had been called from your program.
Note

Target calls are not supported by all debug environments.

Results are displayed in either floating-point format, address format, or in decimal, hexadecimal, or ASCII format depending on the type of variables used in the expression.

SetPC

Submits a $ETREG command to change the contents of a specified register identified by @ followed by the register name.

The Code window scrolls to the specified function and the red box shows the location of the PC.

8.3.4 Scoping to a function

With a list of functions displayed in the Function List you can select an entry and click the Scope+Close button to scope to that function and so adjust the context. This displays a confirmation message in the Output pane.

If the file is not already open in the File Editor pane RealView Debugger opens it automatically, the window scrolls to the right place, and the scope is adjusted. The Function List dialog box closes.

8.3.5 Setting a breakpoint

You can use the Function List dialog box to set a breakpoint on a chosen function:

1. Highlight a function in the display list.
2. Click Break to set a breakpoint.

A cut-down version of the Function browser is also available to set a breakpoint from the Debug menu:

1. Click on the location of your breakpoint in your code view.
2. Select Debug → Simple Breakpoints → Set from Function/Label list... to display the Function Breakpoint/Profile Selector dialog box.

Because the browser is used only to make a selection, there are no controls for debugging operations.

The Function Breakpoint/Profile Selector does not provide a record of breakpoints already set, that is, when you next open this dialog box existing breakpoints are not checked.
8.3.6 Closing the browser

If you scope to a new module or file, the Function List dialog box closes automatically. Otherwise, click Close to exit the browser without adjusting the scope.
8.4 Browsing variables

Use the Variables browser to examine the different variables used in your program.

To display the Variable List dialog box, shown in Figure 8-4, select **Find → Variable List**... from the Code window main menu.

![The Variable List dialog box](image)

The Variable List dialog box shows all the variables in the current program. By default, the **Sort** check box is unselected and so the variables are given in order of occurrence.

Each entry in the list shows the variable name followed by the program name attached using @, for example:

```
Ch_1_Glob of @dhrystone
```

Including local variables adds the filename, if known, to the list, for example:

```
DHRY_2\Proc_8\Int_Index local of @dhrystone
```

This section describes:
- **Specifying the list**
- **Refining the list** on page 8-12
- **Viewing details of a variable** on page 8-12
- **Closing the browser** on page 8-13.

### 8.4.1 Specifying the list

When you first open the Variable List dialog box, the list entries are determined by the default search entry **ISearch** but you can decide which variables are displayed by applying a search filter.
See *Specifying browser lists* on page 8-14 for details of how to specify the list for the chosen browser.

### 8.4.2 Refining the list

The Variable List dialog box contains check boxes that enable you to refine what is displayed in the list box:

- **Publics**: Displays global or public variables with scope over all parts of the program.
- **Statics**: Displays static variables.
- **Labels**: Displays code labels with scope over the entire function.
- **Locals**: Displays local variables with scope within the current function.

### 8.4.3 Viewing details of a variable

The Variable List dialog box contains seven buttons to display more details about a selected variable or to carry out specific debugging activities.

To use these features, highlight a variable in the display list and then click:

- **Print**: Displays the value, in decimal, of the variable in the current procedure.
- **Print Hex**: Displays the variable value in hexadecimal.
- **Watch**: Click to copy the selected variable into the Watch pane so that you can view the watched data and see how the value changes during program execution.
- **PrintType**: Displays type information for the selected variable. This information is displayed in a style similar to the source language.
- **Info**: Displays type information for the selected variable including name, data type, storage class, and memory location.
- **Print+Close**: Displays the variable value in decimal.

When the PRINTVALUE command has completed, the Variable List dialog box closes and control returns to the Code window.
Break+Close

The behavior of this button depends on your debugging environment. Possible results are:

- Sets a breakpoint at the specified location in the image. The position of the breakpoint is indicated by a red disc. If this software execution break is reached while the program is running, RealView Debugger halts execution of the image. When the breakpoint has been inserted, the Variable List dialog box closes and control returns to the Code window.

- Displays the Set Address/Data Break/Tracepoint dialog box with the location filled in.

See Chapter 4 Working with Breakpoints for details on using with breakpoints.

8.4.4 Closing the browser

Click Close to exit the browser.
8.5 Specifying browser lists

When you first open a browser, the list entries shown in the dialog box are determined by the default search entry ISearch but you can decide what contents are displayed by applying a search filter. To change the search mechanism either:

- click on the drop-down arrow to display the search list and select a search
- highlight the contents, for example ISearch, and press F or I to toggle the contents.

This section describes:
- Specifying a list
- Applying a filter on page 8-15.

8.5.1 Specifying a list

When you change the browser search mechanism to specify a new list, RealView Debugger performs the match against list entry segments and not against the text string as shown in the display list. This applies if you choose to sort the display list (see below) or to apply a filter to limit the search (see Applying a filter on page 8-15 for details).

To specify the search:

ISearch With this selected, the list entries are created using the default search mechanism based on the names of the modules, files, functions, or variables found in your program. Enter a partial name to move the highlight to the first matching occurrence. The search is case insensitive.

Filter Use a filter to limit the search to find only those symbols that match certain criteria, see Applying a filter on page 8-15 for details.

Text entry field Enter here the filter to be used in the search and then press Enter. The filter enables you to search using Sun Solaris rules as with ls. You can enter a list of search rules by combining different operators, see Applying a filter on page 8-15 for details of how to apply a search filter.

Sort Select this check box to order the display list in alphabetical order based on the names of the modules, files, functions, or variables found in your program. The sort is case sensitive, that is uppercase and lowercase are treated as identical. This check box is unselected by default.

If you change the default search entry to Filter and then enter a filter to set the display criteria, these settings are maintained when you close, or cancel, the browser.
### Applying a filter

Using a filter enables you to narrow down the search when displaying the list of modules, files, functions, or variables. Table 8-1 shows the metacharacters you can use to specify the filter rule or rules. When entering a filter, characters are case sensitive, for example the filter `*DHRY*` returns a list of five modules but `*dhry*` returns an empty list.

When you have completed the filter, press Enter and the list is refreshed. By repeatedly entering filters, and pressing Enter, you can refine the search to focus on selected modules, files, functions or variables.

<table>
<thead>
<tr>
<th>Metacharacter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>This operator matches any character or number of characters, for example <code>*DHRY*</code> matches <code>MY_DHRYSTONE_H</code> but not <code>Dhrystone_H</code> or <code>MY_DHR</code>.</td>
</tr>
<tr>
<td>?</td>
<td>This operator matches any single character, for example <code>*DHRY_?</code> matches <code>MY_DHRY_A</code> but not <code>MY_DHRY_AB</code> or <code>DHRY_B</code>.</td>
</tr>
<tr>
<td>[...]</td>
<td><code>List</code> operators enable you to define a set of items to use as a filter. The list items must be enclosed by square brackets, for example <code>*[HN]*</code> matches <code>DHRY_H</code> and <code>UNNAMED_1</code> but not <code>STDLIB</code>. An empty list (<code>[]</code>) returns no results.</td>
</tr>
<tr>
<td>^</td>
<td>This operator is used inside a list, to represent a NOT action, for example <code>*_[^2]*</code> matches <code>DHRY_1</code> but not <code>DHRY_2</code>.</td>
</tr>
<tr>
<td>-</td>
<td><code>Range</code> operators enable you to define a range of items to use as a match. The range must be enclosed within square brackets, for example <code>*_[^A-Z]*</code> matches <code>DHRY_H</code> but not <code>DHRY_1</code> whereas <code>*_[^A-Z]*</code> matches <code>UNNAMED_1</code> but not <code>DHRY_H</code>.</td>
</tr>
<tr>
<td>-</td>
<td>Used as the first character in a filter, this operator means do not match. For example, <code>*_SAM* - *HOST*</code> means match all names containing the string <code>SAM</code> except those that contain the string <code>HOST</code>.</td>
</tr>
</tbody>
</table>
8.6 Browsing C++ classes

Use the Symbol Browser pane to examine C++ classes in your application program.

To examine C++ classes either:

- Select the menu option View → Pane Views → Symbol Browser from the Code window main menu.
- Move the focus to the chosen pane, for example the Watch pane, click the Pane Content menu, and select Symbol Browser from the available options.

An example display is shown in Figure 8-5.

![Figure 8-5 C++ symbols in the Symbol Browser pane](image)

This section describes:

- Viewing details of a class
- Viewing details of a function on page 8-17.

8.6.1 Viewing details of a class

Colored icons are used to identify different components within a class:

- **Filled stack + arrow**

  This magenta icon indicates a function which is a declared member of the parent class.
Filled stack  The magenta filled stack indicates that a member function of the parent class is both declared and defined. These members are real in that they are called during execution.

Hollow stack  Where magenta stacks are hollow, they indicate that a member function of the parent class is declared but not defined. These members are virtual in that they are not called during execution.

Filled block  The filled blue block indicates a data object that is manipulated by a class function using operators, or methods, defined in the class.

Right-click on a chosen class to display the Class menu. This contains:

Find Class Definition...  Displays the Find in Files dialog box where you can specify the class details and so locate the definition in your source code.  Use this dialog box to locate class definitions when these are not provided by the compiler.  See the chapter describing searching in *RealView Debugger v1.7 Project Management User Guide* for details on using the Find in Files dialog box.

Properties  Displays a text description of the item under the cursor.

### 8.6.2 Viewing details of a function

Right-click on a function to display the Function menu. This contains:

Show Function Definition  Scopes to the selected function. This option is enabled for defined functions identified by a magenta filled stack icon.

Set Break  Sets a breakpoint at the selected function. This option is enabled for defined functions identified by a magenta filled stack icon.

Properties  Displays a text description of the item under the cursor.
8.7 Other routes to the browsers

You can access cut-down browsers from other routes when working with RealView Debugger. These alternative access paths enable you to use the browsers to select a program structure for:

- running a program trace to display the procedure calling chain from the main program to the current procedure
- setting watches to monitor changes in contents of specific locations.

To view a browser this way:

1. Select View → Pane Views → Watch to display the Watch pane.
2. Click on the Pane menu.
3. Select Enter New Expression... to display the expression prompt box.
4. Click on the drop-down arrow to display the options list.
5. Select the required browser, for example <Variable list...>
   Because the browser is used only to make a selection, there are no controls for debugging operations such as PrintType, Break, or Info.
6. Highlight your choice in the browser display list, and click Select.
   This closes the browser and enters the chosen expression into the data field
7. Click Set to confirm this choice and close the prompt box.

Other dialog boxes include a drop-down arrow that enables you to set the required expression using different browsers, for example click the Pane menu from the Memory pane to choose the option Set New Start Address....
Chapter 9
Working with Macros

In RealView® Debugger, a macro is a C-like function that is invoked by entering a single command using the macro name. This section describes how to define macros for use during your debugging session, and how to save and edit your macros. It contains the following sections:

- *About macros* on page 9-2
- *Using macros* on page 9-8
- *Getting more information* on page 9-17.
9.1 About macros

Macros are interpreted C code running on the host with access to target memory and symbols, user-defined debugger symbols, in host or target memory, and debugger functions. Macros can access debugger variables, external windows and programs, and can be attached to breakpoints, aliases, and windows.

A macro can contain:
- a sequence of expressions
- string formatting controls
- statements
- calls to other macros
- predefined macros
- target functions
- debugger commands.

You can define and use macros at any time during a debugging session to use the commands or statements contained in the macro. You call the macro with a single command using the name. The macro definition might contain parameters that you change each time the macro is called.

When a macro is defined, you can use it as:
- a complex command or in an expression
- an attachment to a breakpoint to create complex breakpoint condition testing
- an attachment to a window to display information in it.

Note
After a macro has been loaded into RealView Debugger, the definition is stored in the symbol table. If the symbol table is recreated, for example when an image is loaded with symbols, any macros are automatically deleted. Disconnecting also clears any macros.

This section gives an overview of macros in RealView Debugger. It includes the following sections:
- Properties of macros on page 9-3
- Debugger commands in macros on page 9-3
- Defining macros on page 9-5
- Calling macros on page 9-5
- Macro return values on page 9-6
- Using macros with breakpoints on page 9-6
- Attaching macros on page 9-6
- Stopping macros on page 9-7.
9.1.1 Properties of macros

Macros can:
- have return values
- contain C expressions
- contain certain C statements
- have arguments
- define macro local variables
- use conditional statements
- call other macros and predefined macros
- be used in expressions, where they return values
- reference target variables and registers
- reference user-defined variables, in debugger or target memory
- execute most debugger commands
- be defined in a debugger include file.

Macros cannot:
- be recursive
- define global variables
- define static variables
- define other macros.

9.1.2 Debugger commands in macros

RealView Debugger enables you to enter debugger commands on the command line, or when using the headless debugger, for example PRINTF or SETMEM.

You can define a macro that contains a sequence of debugger commands. When used in this way, the command must be enclosed by dollar signs ($), shown in Example 9-1.

**Example 9-1 Using debugger commands in macros**

```c
some_commands
define int registers()
{
    some_commands
    for(macro_index = 0; macro_index < 6; macro_index++)
    {
        macro_bin_str[macro_index + 8] = macro_cpsr_mode[macro_index + (6+macro_cpsr_key)];
    }
```
Macros containing commands are similar to command files and can be used for setting up complex initialization conditions. These macros are executed by entering the macro name and any parameters on the RealView Debugger command line.

The following commands cannot be used inside a macro:
- ADD
- DEFINE (unless it is the macro definition itself)
- DELETE
- HELP
- HOST
- INCLUDE
- QUIT.

Because macros can return a value, they can also be used in expressions. When the macro executes, the return value is used according to the return type.

For full details on all CLI commands, see RealView Debugger v1.7 Command Line Reference Guide.

Attaching macros

Macros can also be invoked as actions associated with:
- a window
- a breakpoint
- deferred commands, for example BGLOBAL.

In this case, execution-type commands cannot be used inside a macro:
- GO
- GOSTEP
- STEPLINE, STEPO
- STEPINSTR, STEPOINSTR.
If you require a conditional breakpoint that performs an action and then continues program execution, you must use the breakpoint `continue` qualifier, or return 1 from the macro call, instead of the `GO` command.

### 9.1.3 Defining macros

You can define a macro outside RealView Debugger using an editor and load the macro definition file in with the `INCLUDE` command. The number of macros that can be defined is limited only by the available memory on your workstation.

You can also create, edit, save, and delete macros from the Code window using the `Debug` menu, as demonstrated in `Using macros` on page 9-8.

---

**Note**

After a macro has been loaded into RealView Debugger, the definition is stored in the symbol table. If the symbol table is recreated, for example when an image is loaded with symbols, any macros are automatically deleted. Disconnecting also clears any macros.

---

### 9.1.4 Calling macros

Macros are called from the RealView Debugger command line. The call consists of the macro name followed by a set of parentheses containing the macro arguments, separated by commas.

Macro names are case sensitive and must be entered as shown in the definition. The macro arguments are converted to the types specified in the macro definition. If RealView Debugger cannot convert the arguments it generates an error message.

Examples of macro calls are:

- `mytext(var)` Calls the macro named `mytext` with the argument `var`.
- `count(7)` Calls the macro named `count` with the parameter `7`.

You can define a macro with a name that is identical to a command or keyword used by RealView Debugger. You can then use the macro name in an expression and submit it on the command line where it is interpreted correctly.

If, however, you submit the macro name as a command, RealView Debugger cannot identify it as a macro. To overcome this, use the prefix `MACRO` when entering macro names that might conflict with debugger keywords or command names:

```
MACRO macro_name()
```
Macros take higher precedence than target functions. If a target function and a macro have the same name, the macro is executed unless the target function is qualified. For example, `strcpy` is a predefined debugger macro, while `PROG\strcpy` is a function within the module `PROG`. The predefined macro is referenced as `strcpy(dest,src)`, while `PROG\strcpy(dest,src)` refers to the function within `PROG`.

### 9.1.5 Macro return values

You can use macro return values to control what action RealView Debugger takes when a conditional breakpoint is triggered. If the macro returns a true value, that is nonzero, RealView Debugger continues program execution. If a macro returns a false value, that is zero, RealView Debugger stops program execution.

The type of the macro return value is specified by `return_type` when you define the macro. If `return_type` is not specified, then type `int` is assumed.

### 9.1.6 Using macros with breakpoints

When you set a breakpoint, you can also associate a macro with that breakpoint for complex break conditions. You can also attach predefined macros to breakpoints, for example by using the context menu option `Set BreakIf...` from the `Src` tab.

In this way you can test your program variables and decide whether execution stops or continues after the breakpoint has been triggered. When you have attached a macro to a breakpoint it can be executed every time the breakpoint is triggered.

You can use conditional statements in your macro to change the execution path when the breakpoint is triggered depending on variables on the debug target system or on the host. This enables you to control program execution during your debugging session or when there is no user intervention.

You can also use high-level expressions in macros. Combining these conditional statements and expressions enables you to patch your source program.

Breakpoint macros can be used to fill out stubs, such as I/O handling, and also to simulate complex hardware.

For an example of attaching a macro to a breakpoint and using it to control program execution see *Attaching macros to breakpoints* on page 4-26.

### 9.1.7 Attaching macros

In addition to breakpoints, you can attach macros to:

- aliases, for example `ALIAS my_alias=my_macro()`
- windows, for example `VMACRO 250, my_macro()`
9.1.8 Stopping macros

When macros are run as commands they are queued for execution just like any other debugger command when your program is executing. Click the Command cancel toolbar button to cancel the last command entered onto the queue. This can be used to stop any macro that is running. This does not take effect until the previous command has completed and so any effects might be delayed.

Click the Stop Execution button to stop a macro that is attached to a breakpoint.
9.2 Using macros

This section shows you how to start using macros by working through an example. It contains the following sections:

- Creating a macro
- Viewing a macro on page 9-10
- Testing a macro on page 9-10
- Editing a macro on page 9-11
- Copying a macro on page 9-13
- Deleting a macro on page 9-13
- Calling a macro on page 9-14.

9.2.1 Creating a macro

To create a macros from within RealView Debugger, you can use the Add/Edit Macros dialog box. It is not possible to create macros using CLI commands on the command line.

Complete the following steps to create a macro for use with a debug target:

1. Connect to your target and load an image, for example dhrystone.axf.
2. Select Edit → Editing Controls → Show Line Numbers to display line numbers. This is not necessary but might help you to follow the examples.
3. Select Debug → Add/Edit Debugger Macros... to display the Add/Edit Macros dialog box.
   - When you first open this dialog box, the Existing Macros display list is empty because no macros have been defined or loaded into RealView Debugger.
   - The Macro Entry Area gives advice on how to use the buttons, New, Show, and Copy. This area shows the definition of the macro when it has been created.
4. Click New to create the macro. This inserts the default name int Macro() in the Name data entry field and inserts {} in the Macro Entry Area ready for editing.
5. Edit the default macro name so that it shows int tutorial(var1).
6. Enter the macro contents to show:
   ```c
   int var1;
   {
     printf "value=%d\n",var1;
   }
   ```
When creating a macro, variables must be declared at the start of the macro definition. This also applies to macros that you create using a text editor.

The Add/Edit Macros dialog looks like the example shown in Figure 9-1.

![Image of Add/Edit Macros dialog](image)

**Figure 9-1 Creating a macro**

The macro uses the `PRINTF` command and so the command must be enclosed by dollar signs ($), shown in the Macro Entry Area.

7. Click **Update** to pass the macro definition to RealView Debugger, where it is stored in the symbol table. This adds the new macro to the Existing Macros display list.

   If there are any errors in the macro text, you are notified when you try to pass the macro to RealView Debugger.

   **Note**

   If the symbol table is recreated, for example when an image is loaded with symbols or if you disconnect, any macros are automatically deleted.

8. View the Output pane message:
   ```
   def int tutorial(var1)
   ```

9. Click **Save...** to display the Select file to save/append into dialog box where you can choose where to save the new macro definition file for later re-use.

   The macro name has been entered automatically with the extension `.inc`.

10. Save the new macro, with the default name, in your `\home` directory.

    If the chosen file already exists, RealView Debugger displays a warning message and gives you the option to append the new file or to overwrite the existing contents.
11. Click **Close** to close the Add/Edit Macros dialog box and return to the Code window.

### 9.2.2 Viewing a macro

View the contents of a macro using:
- **the File Editor pane**
- **a standalone editor window or an external text editor**
- **the `SHOW` command.**

Viewing a macro using any of these methods differs from viewing the macro contents in the Add/Edit Macros dialog box:
- The **Macro Entry Area**, in the dialog box, does not show the macro `DEFINE` command or the terminator (a period used as the first and only character on the last line).
- The `SHOW` command does not display the terminator.

**Note**
If you are using a predefined macro, you cannot use the `SHOW` command to view the definition.

### 9.2.3 Testing a macro

Test the macro you have just created by executing the loaded image and then calling the macro from the command line:

1. Click on the **Src** tab to view the source file `dhry_1.c`.
2. Set a simple breakpoint by double-clicking on line 150.
3. Click **Go** to start execution. When asked for the number of runs, use a small number, for example 5000.
4. When the program reaches the breakpoint, enter the following command and press Enter:
   ```
   tutorial(Int_2_Loc)
   ```
   This displays the current value of the variable `Int_2_Loc` in the Output pane.
5. Step through the program a few more times using the macro to monitor the variable. You can use the up arrow to step back through the commands already submitted on the command line.
9.2.4 Editing a macro

You can edit the macro that you have just created and retest it to verify the changes:

1. Select `Debug → Add/Edit Debugger Macros...` to display the Add/Edit Macros dialog. The Existing Macros display list shows the tutorial macro you just created and it is highlighted.

2. Click Show to see the contents of the macro.

3. Change the macro name to read:
   ```
   int tutorial(var1,var2,var3)
   ```

4. In the Macro Entry Area change the variable definition to read:
   ```
   int var1;
   int var2;
   int var3;
   ```

5. In the Macro Entry Area change the body of the macro to read:
   ```
   var1=var1++;
   $fprintf 250,  "value=%d\n",var1$;
   $fprintf 250,  "value=%d\n",var2$;
   $fprintf 250,  "value=%d\n",var3$;
   ```
   This change increments var1 and displays the output of the macro in a window instead of the Output pane.

6. In the Macro Entry Area add this line at the end of the macro:
   ```
   return(var1);
   ```
   This change causes the macro to return the value of the variable. If the value is True, that is nonzero, then RealView Debugger continues program execution after reporting the result. If the value returned is False, that is zero, then execution stops.

The macro now looks like the one shown in Figure 9-2 on page 9-12.
7. Click Update to pass the macro definition to RealView Debugger.

8. View the Output pane message:
   def int tutorial(var1, var2, var3)

9. Click Save to save the updated macro in the same location. This generates a
   prompt to enable you to Append or Replace the existing file. Click No to replace
   the existing tutorial.inc.

10. Click Close to close the Add/Edit Macros dialog.

Test the macro you have just created by executing the image and then calling the macro
from the command line:

1. Select File → Reload Image to Target to reload the image to your debug target.

2. Enter this command:
   VOPEN 250
   This opens a window ready to display the results returned from the macro.

3. Click Go to start execution. When asked for the number of runs, use a small
   number, for example 5000.

4. When the program reaches the breakpoint, enter the command:
   tutorial(Int_1_Loc, Int_2_Loc, Int_3_Loc)
   on the command line of the Code window and press the Enter key. This displays
   the current value of the variables in the window.
5. Step through the program a few more times using the macro to monitor the variables. You can use the up arrow to step back through the commands already submitted on the command line.

Remember to save your macros before you exit RealView Debugger. There is no warning if any macro definitions are unsaved.

### 9.2.5 Copying a macro

You can use an existing macro to form the basis of a new macro:

1. Select **Debug → Add/Edit Debugger Macros...** to display the Add/Edit Macros dialog and so edit the macro. The Existing Macros display list shows the tutorial macro you just created and it is highlighted.

2. Click **Show** to see the contents of the macro.

3. Click **Copy**. This automatically changes the Name field to show `int tutorial1(var1,var2,var3)`. Subsequent copies are called `int tutorial2..`, `int tutorial3..`, and so on. You can change the default name to your own choice.

4. In the Macro Entry Area change the body of the macro as required.

5. Click **Update** to pass the macro definition to RealView Debugger.

6. View the Output pane message, assuming that you do not change the default name:

   ```
   def int tutorial1(var1,var2,var3)
   ```

7. Click **Save** to save the updated macro in the usual way.

8. Click **Close** to close the Add/Edit Macros dialog.

The number of macros that can be defined is limited only by the available memory on your workstation.

### 9.2.6 Deleting a macro

With a group of macros shown in the Existing Macros display list, you can highlight selected macros and click **Delete** to unload them from RealView Debugger. This does not delete the files themselves if they have been saved to disk. You can only delete one macro at a time in this way.

You can also delete a macro, and all associated symbols, using the **DELETE** command.
9.2.7 Calling a macro

When you first start RealView Debugger, any macros you created have been unloaded from RealView Debugger.

After a macro has been loaded into RealView Debugger, the definition is stored in the symbol table. If the symbol table is recreated, for example when an image is loaded with symbols, any macros are automatically deleted.

Note

Reloading an image with all associated symbols also deletes any macros.

To load, or reload, macros into RealView Debugger:

1. Select Debug from the main menu to display the Debug menu.
2. Select Include Commands from File... to display the Select File to Include Commands from dialog box.
3. Highlight the required .inc file and then click Open. This loads the selected macro into RealView Debugger.
   If there is an error in the .inc file, an error message is generated in the Output pane and the macro is undefined.
4. Select Debug → Add/Edit Debugger Macros... to display the Add/Edit Macros dialog box where your macro is now shown in the Existing Macros display list.

You can load in several macros in this way ready for use in your debugging session. When a macro is displayed in the Add/Edit Macros dialog box, it can be changed as described previously and re-used, and resaved if required.

Loading macros on connection

You can load one or more macros automatically when you connect to a given target. Do this by specifying the macros to be loaded using the Connection Properties window. Enter the calling command using the Commands setting in the Advanced Information block, shown in Figure 9-3 on page 9-15.
If RealView Debugger cannot locate one of the specified files, it displays a message box and gives you the option to abort the connection.

For full details on how to specify these commands using the Connection Properties window, see the chapter describing how to configure custom connections in *RealView Debugger v1.7 Target Configuration Guide*.

### Loading macros from a project

You can load one or more macros automatically when you open a project that binds to a connection. Do this by specifying the macros to be loaded using the Project Properties window. Enter the calling command using the Open_conn setting in the Command_Open_Close group, shown in Figure 9-4.
If RealView Debugger cannot locate one of the specified files, it displays a message box. The project then opens and binds as normal.

This might be useful if combined with the Open_load setting in the SETTINGS group, for example to load the associated image and all symbols when the project binds.

For full details on how to specify these commands using the Project Properties window, see the chapter describing how to customize your projects in RealView Debugger v1.7 Project Management User Guide.
9.3 Getting more information

See RealView Debugger v1.7 Command Line Reference Guide for more information on how to use macros:

- see the chapter describing working with the CLI for details on writing your own scripts
- see the chapter describing RealView Debugger commands for full details on using the DEFINE command
- see the chapter describing predefined macros for full details on predefined and user interaction macros in RealView Debugger.
Chapter 10
Configuring Workspace Settings

This chapter explains how to use workspaces in RealView® Debugger, and describes how to configure your workspace settings. Read this chapter in conjunction with Appendix A Workspace Settings Reference that contains a detailed description of the workspace settings.

This chapter contains the following sections:

- Using workspaces on page 10-2
- Viewing workspace settings on page 10-8
10.1 Using workspaces

RealView Debugger uses a workspace to define:

- connection information
- a list of projects to open when the next session starts
- debugger behavior
- windows (sizes and positions) and their attachment
- window contents and panes
- user-defined editor settings and view options.

It is not compulsory to use a workspace when working with RealView Debugger. However, using a workspace enables you to maintain persistence between debugging sessions.

Working without a workspace might be useful to debug an executable file from another developer or for compatibility with other tools. This means that some persistence details are not available.

This section describes workspaces in RealView Debugger. It includes:

- Initializing the workspace
- Workspace menu on page 10-3
- Settings options on page 10-4
- Opening workspaces on page 10-4
- Closing workspaces on page 10-5
- Projects in workspaces on page 10-6
- Creating an empty workspace on page 10-7.

For details on setting up multiple Code windows and attaching to different debug targets, see the multiprocessing chapter in RealView Debugger v1.7 Extensions User Guide.

10.1.1 Initializing the workspace

The first time you run RealView Debugger after installation, it creates a default workspace to define your initial working environment. Two files are created in your RealView Debugger home directory to store settings:

rvdebug.aws Contains workspace-specific settings that apply to the current workspace.

rvdebug.ini Contains global configuration options that apply to all workspaces, or are used when working without a workspace.
By default, at the end of your session, the .aws file is updated to save the current workspace and is used when you start your next session. The global configuration file is updated when it is edited or at the end of your session if you are working without a workspace.

**Start-up options**

You can start RealView Debugger with a specified workspace from the command line, or by using a desktop shortcut, for example:

```
rvdebug.exe -aws="D:\ARM\RVD\home\my_user_name\myws_rvdebug.aws"
```

You can start a debugging session without a workspace, for example:

```
rvdebug.exe -aws=-
```

**10.1.2 Workspace menu**

In a debugging session you can:

- edit your workspace settings and resave them ready for the next session
- set up specific workspaces containing custom settings for use during selected debugging sessions or with particular application programs
- switch between workspaces, without exiting the debugger, to continue previous debugging sessions
- close the current workspace and continue the session without a workspace.

To manage your current workspace, select **File → Workspace** from the Code window menu to display the **Workspace** menu:

- **Open Workspace**...
  
  Displays a dialog box where you can locate a workspace to open, see *Opening workspaces* on page 10-4 for details.

  If you are already using a workspace, select this option to close the current workspace before the new workspace opens, see *Closing workspaces* on page 10-5 for details.

- **Save Workspace**

  Saves the current workspace to disk. This is useful if you have made changes since your debugging session began. The workspace file, for example rvdebug.aws, is saved with the same name and the workspace backup file is updated.
Save As Workspace...

Saves the current workspace to disk using a new name. This is useful if you have made changes since your debugging session began and want to save this new setup in a new workspace. This displays a dialog box where you can specify the new filename, for example test_workspace.aws. The newly-specified workspace becomes the current workspace.

Close Workspace

Closes the current workspace. After the workspace closes, RealView Debugger displays a list box so you can close any open objects. See Closing workspaces on page 10-5 for more details.

10.1.3 Settings options

RealView Debugger enables you to specify that settings are saved automatically at the end of the debugging session. If selected, settings are saved in your start-up file, using the .sav extension by default, for use next time and the current workspace file is updated.

At the bottom of the Workspace menu are your current settings options:

Save Settings on Exit

Selected by default, this option saves selected user-configured settings and the current workspace when you exit RealView Debugger. This enables you to start your next debugging session in the same state.

Same Workspace on Startup

Selected by default, this option saves the current workspace pathname in your .sav file so that the same workspace is used at the next start-up. You can unselect this option so that the current workspace is not opened by default when you next start RealView Debugger. Unless you specify a workspace on the command line, RealView Debugger then runs without a workspace.

10.1.4 Opening workspaces

If you open a new workspace, RealView Debugger adds the objects specified in the new workspace to all existing objects. This usually means that at least one more Code window opens on your desktop.

If you open a new workspace, you might see many new Code windows on your desktop. To avoid this, close open windows before opening the new workspace, see Closing workspaces on page 10-5 for details.
When you open a new workspace, it might contain settings that override the current configuration. Where there is a conflict, a warning message is displayed and the new workspace settings are used.

### 10.1.5 Closing workspaces

You can explicitly close your current workspace, either:

- select **File → Workspace → Close Workspace**
- select **File → Workspace → Open Workspace...** to close the current workspace before the new one opens.

If you close your current workspace, the following applies:

- The contents of the default Code window do not change.
- If there are open objects, these do not change (see Closing objects for details).
- Any open objects are saved in the workspace before it closes so that they can be re-used when it next opens.
- If there are no open objects, the current workspace closes immediately.
- If the Workspace Options window is open, this closes automatically before the current workspace closes. If you have changed any workspace settings, these are saved.
- If the Options window is open, this is not affected when the workspace closes.

### Closing objects

If you close a workspace, RealView Debugger enables you to close any open objects. This might be useful to restore a clean desktop for the session, or before you open a new workspace.

After the current workspace closes, RealView Debugger displays a list selection box where you can specify the open objects you want to close, shown in Figure 10-1 on page 10-6.

--- **Note**

The Close Open Objects selection box is not displayed if there are no open objects.
Configuring Workspace Settings

The display list shows the open objects, that is:

- connections to debug targets
- any windows open in addition to the default Code window
- open projects including user-defined projects and auto-projects (see Projects in workspaces for details).

Each entry has an associated check box that is ticked by default. Select the check box to unselect objects. The list selection box contains the controls:

- **OK**: Click this button to close selected objects and then close the selection box.
- **Cancel**: Click this button to ignore the status of any check boxes in the list and close the selection box. Use Cancel to maintain all open objects.
- **Help**: Click this button to display the online help.

You can use the Close icon to close the selection box. This is the same as clicking Cancel.

10.1.6 Projects in workspaces

RealView Debugger saves a project load list when the current workspace closes. This is a list of open projects maintained when the debugger starts with this workspace or when you open this workspace in a session. This list includes user-defined projects and any auto-projects where you have saved the settings.

Where you are using a project load list, you must be aware of the following when the workspace opens:

- Where a project was bound to a connection when the workspace closed, binding details are saved and this is maintained when the connection is restored.
Project binding details saved in the workspace take precedence. This applies even where the load list opens an autobound project that is unbound, that is where you have defined a Specific_device setting.

Where there is no connection, the order in which projects open defines the active project.

If you are already running a debugging session and you open a new workspace, the project environment might change depending on the project load list (if any). RealView Debugger forces the project binding as defined in the workspace even if this means unbinding open projects. This is true even if the open projects include an autobound project, that is a project where you have defined a Specific_device setting.

If the workspace opens with no saved binding details, the current project environment does not change. This is true even if the project load list contains a project where you have defined a Specific_device setting.

--- Note ---
There is no warning when the project environment changes as a result of opening a workspace into a debugging session.

If you are licensed to work with multiple projects in multiprocessor debugging mode, the workspace restores the project environment based on:

- your connections
- the order in which projects open
- saved project binding details
- open windows and their attachment.

See RealView Debugger v1.7 Project Management User Guide for full details on working with projects and project binding.

10.1.7 Creating an empty workspace

You can create a blank workspace settings file at any point during your debugging session. To do this, select File → New → Workspace... from the Code window main menu. This displays the New Workspace dialog box.

Use this to create an empty file, for example New_workspace.aws, in the new location. This becomes the current workspace. You must save settings to this new workspace settings file if you want it to be available at the next start-up.

If you are already working with a workspace this closes and then the new workspace opens ready for you to use. This does not override the current configuration.
10.2 Viewing workspace settings

RealView Debugger provides the Workspace Options window to enable you to examine, and change, workspace settings. Use this interface to see the contents of the .aws file and the .ini file.

There are descriptions of the general layout and controls of the RealView Debugger Settings windows in the RealView Debugger online help topic Changing Settings. This chapter assumes familiarity with the procedures documented in that topic.

This section contains:
- Using Settings windows
- Options window
- Workspace Options window on page 10-9
- Groups and settings on page 10-11.

10.2.1 Using Settings windows

Select Tools from the Code window main menu to display the Tools menu and access your current workspace settings or global configuration options:

Workspace Options... Displays the Workspace Options window where you can view the current workspace settings or make changes.

Options... Displays the Options window where you can view the global configuration options, used by the current workspace, or make changes. You have access to the global options when you are working without a workspace.

10.2.2 Options window

The Options window enables you to examine, and change, your current global configuration options. These settings are saved in the file rvdebug.ini and are included when the default workspace opens for the first time.

If you are working without a workspace, use this window to make the changes described in the rest of this chapter.
10.2.3 Workspace Options window

The Workspace Options window enables you to examine your current workspace settings and edit these settings to change the workspace or to create your own workspace files. The first time RealView Debugger opens the default workspace file, rvdebug.aws, the Workspace Options window contains only the start-up settings, shown in Figure 10-2.

The main interface components of this window are:

**Main menu** This contains:
- **File** Displays the File menu where you can save the workspace file after you have made changes.
- **View** Displays the View menu to toggle the display to show all the settings or only those that have been edited.
- **Help** Displays the online Help menu.

**Save icon** Click this icon to save the workspace settings file to disk. The name of the current file is shown as the first entry in the left pane.

**Description** This field displays a one-line description about an entry selected in the panes below.
List of Entries and Settings Values

The left pane of the Workspace Options window, the List of Entries pane, shows workspace entries as a hierarchical tree with node controls (see Figure 10-3). Groups of settings are associated with an icon to explain their function:

- **Red disk**  This is a container disk file.
  RealView Debugger uses this, for example, to specify an include file.

- **Yellow folder**  This is a parent group containing other groups (rules pages) and/or entries.

- **Rules page**  A rules page is a container for settings values that you can change in the right pane. When unselected, the pencil disappears (see Figure 10-3). This icon only appears in the left pane.

When you close down RealView Debugger, your workspace settings file is updated with the current configuration, for example projects, connections, and open windows. An asterisk (*) is placed at the front of an entry to show that it has changed from the default or was created by RealView Debugger. Figure 10-3 shows an example settings file that specifies a multiprocessor debugging session.

If you click on an entry in the left pane, a red box is drawn around it and the Description field is updated. At the same time, the right pane, the Settings Values pane, is updated to show the contents of the highlighted group.
### Note

See the RealView Debugger online help topic *Changing Settings* for details on all the entries in the Workspace Options window.

## 10.2.4 Groups and settings

Workspace settings, in the left pane, are grouped according to their function:

**Workspace file**

This is the current workspace settings file. Click on this entry to see the full pathname in the Description field.

**Global configuration file**

This is the global configuration file, `rvdebug.ini`, included in the current workspace.

You can set up DEBUGGER, CODE, or ALL groups in your workspace settings file or in your global configuration file. RealView Debugger issues a warning if conflicts are detected when the workspace opens and uses settings from the new workspace file.

**DEBUGGER**

These settings govern the behavior of generic actions in the debugger. These controls are used in conjunction with other processor-specific controls.

**CODE**

These settings govern the behavior of all Code windows. They control the display characteristics of windows, including size and position, and any user-defined buttons created on the toolbars (not available in this release).

**ALL**

These settings govern the behavior of the editor, the editor display, and access to source code. The settings in the ALL group are used in conjunction with the settings in the DEBUGGER group and the CODE group and might be overridden by settings in either of these two groups.

**PROJECTS**

This specifies a project load list, that is a project, or projects, opened when the debugger starts with this workspace or when you open this workspace in a session. This list includes user-defined projects and any auto-projects where you have saved the settings.

This group is created automatically when RealView Debugger closes down with open projects, or if you close the current workspace.

**WINDOW**

This is a special group of windows internals maintained by RealView Debugger. An entry is created for each open window. Entries cannot be edited.

You must not delete these entries.
**CONNECT**  When you close down, you have the option to save connection details so that the same connections are used when RealView Debugger starts up with the same workspace.

This is a special group, to specify connections, maintained by RealView Debugger. An entry is created for each connection. Entries cannot be edited.

You must not delete these entries.

For a full description of all the workspace settings you can change see Appendix A *Workspace Settings Reference*. 
10.3 Configuring workspace settings

The following notes apply to changing your workspace settings:

- Settings are applied in the order they are shown in the settings hierarchy in the left pane. This means that settings in the workspace file take priority over global configuration settings if a conflict arises when you open a workspace.

- If you edit the workspace settings, the .aws file is updated when you save the change. This change takes effect in any new Code windows you open in the current session.

- Use the Options window to make changes to global configuration options saved in the rvdebug.ini file.

- If you edit the global configuration options, the .ini file is updated when you save the workspace file. This change takes effect when the workspace next opens.

- Do not change the same setting in the Workspace Options window and the Options window at the same time because the views might not be consistent.

This section describes:

- Restoring settings
- Changing settings on page 10-14
- Copying entries on page 10-15
- Pasting entries on page 10-15
- Cutting entries on page 10-17
- Resetting entries on page 10-17.

10.3.1 Restoring settings

If you have made changes to your workspace settings file, or your global configuration file, and you want to restore the factory settings:

1. Exit RealView Debugger in the normal way.
2. Locate the .aws and the .ini files in your RealView Debugger home directory.
3. Delete both files.
4. Start RealView Debugger and accept the option to create a new .aws file.
10.3.2 Changing settings

This section includes examples of changes that you can make to your current workspace. Select **Tools → Workspace Options...** to display the Workspace Options window to edit the workspace file. This means that changes take effect in the current workspace:

- Configuring the Code window
- Setting up debugger options.

When you have saved a change, select **View → New Code Window** to display a new Code window to see the effect.

**Configuring the Code window**

To change the size of the Code window:

1. Expand the **CODE** group.
2. Expand the **Pos_size** group.
3. Right-click on the default **Num_lines** setting.
4. Select **Edit Value** from the context menu.
5. Use in-place editing to set the value to 0x040.
6. Press Enter to confirm your setting.
7. Save the updated version of the workspace settings file.

--- **Note** ---

To restore the Code window, select the option **Reset to Empty**.

---

**Setting up debugger options**

To change the height of the Output pane:

1. Expand the **DEBUGGER** group.
2. Expand the **Command** group.
3. Right-click on the default **Num_lines** setting.
4. Select **Edit Value** from the context menu.
5. Use in-place editing to set the value to 10.
6. Press Enter to confirm your setting.
7. Save the updated version of the workspace settings file.

--- Note ---
To restore the Code window, select the option Reset to Empty.

### 10.3.3 Copying entries

When you are working in the Workspace Options window, context menus include options to enable you to copy settings so that you can make changes quickly. The options that are available depend on the:

- pane you are working in
- contents of the clipboard
- relationship between what is on the clipboard and the entry under the cursor when you right-click.

The available options are:

**Copy**

Select this option to make a copy of an entry to the clipboard ready for pasting.

This option is always available in the left pane to copy settings groups. When you are working in the right pane, this option depends on the entry under the cursor.

**Make Copy...**

Where permitted, this option enables you to make a copy of the chosen group. A dialog box enables you to define a new name for the copy.

### 10.3.4 Pasting entries

When you are working in the Workspace Options window, context menus include options to enable you to paste settings so that you can make changes quickly. Like the copy options, the paste options that are available depend on the:

- pane you are working in
- contents of the clipboard
- relationship between what is on the clipboard and the entry under the cursor when you right-click.
The available options are:

**Paste Group Into**

This option is only available if you right-click on a settings group, or a container disk file, with a settings group already copied to the clipboard.

This option is usually available in the left pane to paste settings groups into the settings file, or to copy between the workspace settings file and the global configuration file.

When you are working in the right pane, this option depends on the entry under the cursor. This means that you might not be able to paste the contents of the clipboard into the chosen location.

**Paste Rule Here**

Certain settings in the right pane are classed as *rules*. In particular, you can use rules to specify settings for projects when you are working in the Project Properties window. See the chapter describing customizing projects in *RealView Debugger v1.7 Project Management User Guide* for details.

This option is only available if you right-click on a settings group, or a container disk file, with a single rule setting already copied to the clipboard.

This option is available in the left or right pane if you select a settings group that can accept the rule currently on the clipboard. This means that you might not be able to paste the contents of the clipboard into the chosen location.

**Paste as 1st Child**

To see this option, you must right-click on a parent group with a child group already copied to the clipboard.

Use this option, in the left pane, to paste a settings group into a parent group, that is to create a sibling group.

When you are working in the right pane, use this option to paste a child group. The paste only succeeds if the chosen parent group can accept the child. This means that you might not be able to paste the contents of the clipboard into the chosen location.

This option might be replaced by *Paste After*. This depends on the entry under the cursor.

**Paste After**

Use this option, like *Paste as 1st Child*, to paste a settings group into a parent group, that is to create a sibling group.
This option enables you to specify the relationship between the new group and other siblings. This determines the order in which settings are used.

**Paste String** This option is only available in the right pane if you right-click on a settings group, or a container disk file, with a string setting already copied to the clipboard.

The paste only succeeds if you select a setting that can accept the string currently on the clipboard. This means that you might not be able to paste the contents of the clipboard into the chosen location.

**Paste Value** This option is only available in the right pane if you right-click on a settings group, or a container disk file, with a value setting already copied to the clipboard.

The paste only succeeds if you select a setting that can accept the value currently on the clipboard. This means that you might not be able to paste the contents of the clipboard into the chosen location.

### 10.3.5 Cutting entries

When you are working in the Workspace Options window, context menus include the option **Cut** to enable you to mark an entry for deletion. The entry is grayed out until you paste it into a new location.

If you cut another entry before you paste the first entry, the first entry is restored.

If you cut an entry, do not delete it until it has been pasted. If you delete the cut entry, it is no longer available to paste. This also applies to entries that you copy.

Always use a **Delete** option to remove unwanted entries.

### 10.3.6 Resetting entries

An asterisk is placed at the front of any entry that you edit in the workspace settings file to show that it has changed from the default. This also applies to entries maintained by RealView Debugger. You can reset all values using the **File → Reset** option from the window menu. This updates the window with the settings currently saved on disk. You are warned that any changes made since you last saved will be lost.

When you have changed a value, right-click to see the context menu showing the option **Reset to Default**. Select this to change the value to the default and cancel any changes.

Right-click on a group of settings and select **Delete Contents** from the context menu to reset it back to empty. This deletes all the changed settings and restores the defaults. There is no undo.
Configuring Workspace Settings
Appendix A
Workspace Settings Reference

This appendix contains reference details about settings that define the RealView®
Debugger workspace and global configuration options. It contains the following
sections:

- *DEBUGGER* on page A-2
- *CODE* on page A-6
- *ALL* on page A-8.
A.1 DEBUGGER

Settings in this group govern the behavior of generic actions in the debugger. These controls are then used in conjunction with other processor-specific controls.

DEBUGGER contains two second-level groups and a file:

- **Command**
- **Disassembler** on page A-3
- **Board_file** on page A-4.

A.1.1 Command

Settings in this group control the behavior and appearance of the Code window command line and Output pane. Use these to customize the input and output format used in this area.

When RealView Debugger starts, it uses the last-used settings unless overridden by settings in this group. These settings can be overridden dynamically by issuing CLI commands.

Saving changes takes immediate effect or at next start-up.

Table A-1 describes the settings.

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num_lines</td>
<td>The height of the Output pane. The default setting is 5 lines. Values can be entered in hex or decimal, for example 15 or 0x000F.</td>
</tr>
<tr>
<td>Radix_in</td>
<td>This setting specifies the format of number input options at start-up. The default format is decimal. However, you can enter hex numbers, for example <code>ce number=0xABCD</code>. Switching to hex also enables you to enter decimal numbers, for example <code>ce number=01234t</code>.</td>
</tr>
<tr>
<td>Radix_out</td>
<td>This setting specifies number format output options at start-up. The default format is decimal.</td>
</tr>
</tbody>
</table>

Table A-1 Command settings
A.1.2 Disassembler

Settings in this group control how the disassembly view is displayed in the Code window. This can be set for all processors or for specific processors only. The default settings apply to all processors.

When RealView Debugger starts, it uses the last-used settings unless overridden by these settings. These settings can be overridden dynamically by issuing CLI commands. Saving changes takes immediate effect.

--- Note ---
Some processor disassemblers do not support features configured with these settings, and the settings are ignored.

Table A-2 describes the settings.

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbols</td>
<td>When instructions reference direct memory locations, either relative to the PC or as absolute references, the debugger tries to show the symbol at that location. Use this to disable this property.</td>
</tr>
<tr>
<td>Labels</td>
<td>When an instruction has a label associated with the address, the debugger shows it inline. Use this to disable this property.</td>
</tr>
<tr>
<td>Source</td>
<td>By default, high-level source code is interleaved with disassembly code when available. Use this to disable this property.</td>
</tr>
</tbody>
</table>
### Table A-2 Disassembler settings (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Asm_source</strong></td>
<td>By default, assembler source code is interleaved with disassembly code when available. This is isolated from high-level language source code display because the assembly source is usually of less interest in this mode. Not all assemblers produce the information to enable assembly tagging. This setting does not affect what is shown in the <strong>Src</strong> tab in the File Editor pane.</td>
</tr>
<tr>
<td><strong>Source_line_cnt</strong></td>
<td>By default, interleaved display shows eight lines of source code for any instruction. If there are more source lines associated with the instruction, they are not shown. Use this to define how many lines are shown, or to specify that all are shown.</td>
</tr>
<tr>
<td><strong>Stack_syms</strong></td>
<td>Some disassemblers identify frame or stack offset references in the operand fields. Use this to display the corresponding stack-based variable when possible.</td>
</tr>
<tr>
<td><strong>Register_syms</strong></td>
<td>Not available with ARM® tools. Some disassemblers identify register usage in the operand fields. Use this to show the corresponding register-based variable when possible.</td>
</tr>
<tr>
<td><strong>Format</strong></td>
<td>Some disassemblers include alternate format which is processor-specific. By default, RealView Debugger shows the format that is most appropriate for the given processor context. Use this to change the default format. You can also change the format dynamically using the <strong>DISASSEMBLE</strong> command, that is <strong>DISASSEMBLE /D</strong>, for default format, or <strong>DISASSEMBLE /A</strong>, for alternate format When debugging ARM processors, use this to force the disassembler to display 16-bit values, as opposed to showing the appropriate format for the given processor context, that is mixed 16-bit and 32-bit values. Some DSPs use this to differentiate mnemonic and algebraic format.</td>
</tr>
<tr>
<td><strong>Instr_value</strong></td>
<td>In general, disassemblers show instructions as values and as opcodes or operands. Use this to suppress the value display where possible.</td>
</tr>
</tbody>
</table>

### A.1.3 Board_file

Change this setting to specify a different board file for the current session.

Changing this value takes immediate effect. The specified board file is read and the contents used to populate the configuration details.
Resetting the value back to Empty does not take effect until the next time RealView Debugger starts.
A.2 CODE

Settings in this group govern the behavior of all Code windows when running a debugging session. These settings control the display characteristics of windows, their size and position, and any user-defined buttons created on the toolbars (not available in this release).

CODE contains two second-level groups and a settings rule:
- Pos_size
- Button
- Asm_type on page A-7.

A.2.1 Pos_size

Settings in this group control the position on screen and the size of Code windows in lines and characters. Use these to customize the size and position of Code windows in the debugger.

On start-up, RealView Debugger uses the last-used settings unless overridden by these settings. As you open new Code windows in the session, they are controlled by these settings.

Table A-3 describes the settings.

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num_lines</td>
<td>Use this to specify window height as a given number of lines. The default is 0x0200.</td>
</tr>
<tr>
<td>Num_chars</td>
<td>Use this to specify window width as a given number of characters. The default is 0x02A0.</td>
</tr>
<tr>
<td>X_pos</td>
<td>Use this to specify the X position of the top-left corner of the window. The default is 0x010F.</td>
</tr>
<tr>
<td>Y_pos</td>
<td>Use this to specify the Y position of the top-left corner of the window. The default is 0x0066.</td>
</tr>
</tbody>
</table>

A.2.2 Button

Note

These settings are not available in this release.
A.2.3  Asm_type

When an assembler source file opens, RealView Debugger decides what type of processor is in use. However, the processor type is unknown if:

- there are no active connections
- there are no user-defined projects currently open
- there are no auto-projects currently open.

In this case, RealView Debugger does not know the format of instructions and so cannot define source coloring rules. This generates a selection box where you can specify the processor type.

Change this to specify a default processor type on start-up, for example ARM.

Saving a change to this setting takes immediate effect on new assembler source files opened following the update.
A.3 ALL

This group contains three second-level groups:

- Text
- Search on page A-9
- Edit on page A-10.

A.3.1 Text

Settings in this group control the File Editor pane and editor functions within the Code window.

Saving changes might not take effect until the next time RealView Debugger starts, or when a new Code window opens, or when a standalone editor starts.

The Text group contains one third-level group and a series of settings:

**Source_coloring**

These settings control the colors used to identify source tokens. The defaults have been chosen to be easy to read and work well to isolate different program areas. The coloring choices are made relative to the built-in color models.

Table A-4 describes the settings.

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>File_extensions</td>
<td>The standard C/C++ source coloring is auto-enabled based on file extension. Use this to specify a comma-separated list of file extensions that, when loaded, trigger source code coloring.</td>
</tr>
<tr>
<td>Numbers</td>
<td>Use this to specify the color for numbers displayed in the File Editor pane or in a standalone editor window.</td>
</tr>
<tr>
<td>Strings</td>
<td>Use this to specify the color for strings displayed in the File Editor pane or in a standalone editor window.</td>
</tr>
<tr>
<td>Keywords</td>
<td>Use this to specify the color for C/C++ keywords displayed in the File Editor pane or in a standalone editor window.</td>
</tr>
<tr>
<td>Comments</td>
<td>Use this to specify the color for comments displayed in the File Editor pane or in a standalone editor window.</td>
</tr>
</tbody>
</table>
Settings in this group configure editor behavior when working with source files in the File Editor pane or in a standalone editor window. Table A-5 describes these settings.

### Table A-5 Text settings

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Height</td>
<td>Use this to specify the height, in number of lines, for text displayed in the File Editor pane or in a standalone editor window.</td>
</tr>
<tr>
<td>Width</td>
<td>Use this to specify the width, in number of characters, for text displayed in the File Editor pane or in a standalone editor window.</td>
</tr>
<tr>
<td>Src_color_dis</td>
<td>Source coloring is used to make it easier to read source of high-level and low-level languages. All source coloring can be disabled in which case all text will be the same color (usually black). By default, source coloring is enabled, that is this setting is False.</td>
</tr>
</tbody>
</table>

### A.3.2 Search

Settings in this group control the searching behavior when working with source files in the File Editor pane.

These settings can be overridden dynamically using the menus and toggles in the File Editor.
Table A-6 describes these settings.

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Use this to specify the search direction. The default is to search forwards, that is, from the top to the bottom of the file.</td>
</tr>
<tr>
<td>Wrap</td>
<td>Use this to specify search behavior when the end of file is reached. The default is to wrap during a search, that is, to search to the end of the file and then to start again at the top until the starting point is reached.</td>
</tr>
<tr>
<td>Sensitive</td>
<td>Use this to specify whether uppercase and lowercase characters are treated as identical in searches. By default, searches are case-sensitive.</td>
</tr>
<tr>
<td>Regexp</td>
<td>When set to True, full grep-style regular expressions are used in searches. The default is false, not enabled. When working in vi command mode, regular expressions are enabled by default, unless disabled using the :set command.</td>
</tr>
<tr>
<td>Fail</td>
<td>Use this to specify editor behavior when a search fails. Set to dialog by default, you can change this to flash. When working in vi mode, a message is displayed in the vi command line when no match is found.</td>
</tr>
</tbody>
</table>

### A.3.3 Edit

Settings in this group control general editor behavior when working with source files in the File Editor pane. These settings can also be used to control the operation of a standalone editor if specified for use outside RealView Debugger.

These settings can be overridden dynamically using the menus and toggles in the File Editor.

The **Edit** group contains three third-level groups and a series of settings:

#### Backup

These settings control the backup behavior when working with source files in the File Editor pane.
Table A-7 describes these settings.

### Table A-7 Backup settings

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable</td>
<td>By default, a backup file is created when a file is edited. This provides a useful safety feature. Use this to disable this feature if required.</td>
</tr>
<tr>
<td>Backup_dir</td>
<td>By default, backup files are saved in the same directory as the original file. Use this to specify a pathname to a new location, for example to keep all backup files in one special directory.</td>
</tr>
<tr>
<td>Backup_ext</td>
<td>By default, backup files are saved with the .bak extension appended to the original filename.</td>
</tr>
</tbody>
</table>

**Tab_conv**

Settings in this group control the display behavior when working with source files in the File Editor pane. These settings are used to handle tabs and spaces.

Tabs are permitted in files and are left untouched, by default. Use these settings to convert tabs to spaces when writing to the file, that is saving, and to convert spaces to tabs when reading the file.

Spaces are not converted to tabs inside “” and “” quoting blocks on a line.

Table A-8 describes these settings.

### Table A-8 Tab_conv settings

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tabs_to_spaces</td>
<td>Converts tabs to spaces when the file is saved.</td>
</tr>
<tr>
<td>Spaces_to_tabs</td>
<td>Converts spaces to tabs when the file is read.</td>
</tr>
<tr>
<td>To_spaces_ext</td>
<td>Use this to specify file extensions where tab conversions take place. Specify a list separated by semi-colons (;)</td>
</tr>
<tr>
<td>To_tabs_ext</td>
<td>Use this to specify file extensions where space conversions take place. Specify a list separated by semi-colons (;).</td>
</tr>
</tbody>
</table>

**Src_ctrl**

Settings in this group control source control access tools when working in the File Editor pane and in the debugger. The editor attempts to detect any source control system in use, where possible, but you might have to specify it before RealView Debugger can use it.
Use these settings to specify complete source control commands, and to override commands for known systems.

The **Src_ctrl** group contains a low-level group and a series of settings:

**Cmds**  
Use these to specify source control commands for use with RealView Debugger.

**Src_ctrl settings**  
Use these to specify the source control system to be used with RealView Debugger to control access to files.

For full details on these settings see the chapter describing version control in *RealView Debugger v1.7 Project Management User Guide*.

**Edit settings**

Settings in this group configure editor behavior when working with source files in the File Editor pane.

Table A-9 describes these settings.

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drag_drop_dis</td>
<td>Use this to disable drag-and-drop editing when working in the File Editor pane.</td>
</tr>
<tr>
<td>Vi</td>
<td>Running the editor in vi mode enables you to access all the vi commands and most ex commands. You can configure the debugger to start the File Editor in vi mode using this setting. When in insert mode, all Common User Access (CUA) editing features are available. You can also enable this option from the File Editor pane menu bar.</td>
</tr>
<tr>
<td>Indent</td>
<td>Use this to set indenting so that a specified number of spaces are inserted as you open a new line. By default, auto-indent inserts the same number of spaces as on the previous line. If the previous line is a left curly bracket ({), the shift is increased. If the previous line is a right curly bracket (}), shift spaces are subtracted.</td>
</tr>
<tr>
<td>Undo</td>
<td>Use this to specify the levels of undo and redo. By default, this is set to 64.</td>
</tr>
<tr>
<td>Tab</td>
<td>Use this to specify the size of TAB settings when working in the File Editor. By default, this is set to 8. Use a value between 1 and 16.</td>
</tr>
<tr>
<td>Shift</td>
<td>Use this to specify the size of shift spaces as used in the Indent rule and accessed through the File Editor menu options. By default, this is set to 2. Use a value between 2 and 32.</td>
</tr>
</tbody>
</table>
### Table A-9 Edit settings (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line_number</td>
<td>By default, line numbering is disabled in the debugger and the File Editor. Use this to change the editor default to show line numbers at start-up.</td>
</tr>
<tr>
<td>No_tooltip</td>
<td>By default, tooltip evaluation of variables and registers is enabled. Change this setting to True to disable this feature.</td>
</tr>
</tbody>
</table>
| Timer       | During file editing, the editor periodically checks to see if another tool has edited or deleted the files being tested. A warning is shown if an update is detected.  
Use this to specify the number of seconds between checks. The default is 60 seconds. Use values greater than 30 seconds. Set to -1 to disable this feature. |
| Tool_save   | When performing a build, you are prompted to resave any files that have been edited. Use this to specify automatic resaving of changed files at build time to ensure that your latest sources are included. You can also set a no-save, no-ask value. |
| Startup     | The default start-up file, that is rvdebug.sav in your home directory, contains a list of previously edited files and information from previous debugging or editing sessions. This enables historical information to be separated from your current session.  
Use this to specify a different start-up file, in a new location. Set this to - (dash) to specify that no start-up file is used. |
| Template    | During file editing, you can use templates to speed up code development. The template file contains templates that you can use or edit as required.  
By default, the file is named rvdebug.tpl and is saved in your home directory, or in the default settings directory \etc. Use this to change this pathname. |
| Restore_state | This setting applies only to a standalone editor. Use this to start the standalone editor in the same state it was in when you last exited. |
Appendix B
RealView Debugger on Sun Solaris and Red Hat Linux

This appendix provides supplementary information for developers using RealView® Debugger on Sun Solaris and Red Hat Linux. It contains the following sections:

- About this Appendix on page B-2
- Getting more information on page B-3
- Changes to target configuration details on page B-4
- Changes to GUI and general user information on page B-7.
B.1 About this Appendix

This appendix describes features that are specific to using RealView Debugger v1.7 on Sun Solaris and Red Hat Linux, and contains corrections and additions to the documentation suite. It covers:

- RealView Debugger Essentials Guide
- RealView Debugger User Guide
- RealView Debugger Project Management User Guide
- RealView Debugger Target Configuration Guide
- RealView Debugger Command Line Reference Guide
- RealView Debugger Extensions User Guide.
B.2 Getting more information

This section describes how to get more information on RealView Debugger:

- Online resources
- Feedback on RealView Debugger.

B.2.1 Online resources

ARM® Limited provides a range of services to support developers using RealView Debugger. Among the downloads available are:

- enhanced support for different hardware platforms through technical information and board description files
- product Release Notes
- updates to documentation
- Frequently Asked Questions.

You can access these resources from the ARM web site at http://www.arm.com.

B.2.2 Feedback on RealView Debugger

If you have any problems with RealView Debugger, submit a Software Problem Report (SPR):

1. Select Help → About... from the RealView Debugger main menu.
2. Click File an SPR on the About Box Information dialog box.
3. Complete all sections of the Software Problem Report.
4. To get a rapid and useful response, give:
   a small standalone sample of code that reproduces the problem, if applicable
   a clear explanation of what you expected to happen, and what actually happened
   the commands you used, including any command-line options
   sample output illustrating the problem.
5. Email the report to your supplier.
B.3 Changes to target configuration details

This section describes additions to RealView Debugger v1.7 Target Configuration Guide:

- Default configuration files
- Target configuration entries.

B.3.1 Default configuration files

Because you are using RealView Connection Broker connections to RealView ARMulator® ISS (RVISS), the RDI configuration files, described in the documentation suite, are not installed when running RealView Debugger on Sun Solaris or Red Hat Linux, for example *.rbe or *.cnf files. Other configuration files are installed in:

```
install_directory/RVD/Core/1.7/build/solaris-sparc/etc
```

These files are:

- board file, for example rvdebug.brd
- JTAG files, for example arm.jtg
- Board/Chip definition files, for example CM940T.bcd.

For full details on these files and the settings they contain, see the chapter describing configuring custom targets in RealView Debugger v1.7 Target Configuration Guide.

B.3.2 Target configuration entries

RealView Connection Broker enables you to connect to local or remote simulators, such as RVISS. You can also connect to local or remote On-Chip Debugging (OCD) based emulators such as Multi-ICE direct connect.

This means that target configuration entries related to remote connections are fully supported by RealView Debugger, for example the Remote group, shown in Figure B-1 on page B-5.
Figure B-1 Remote entries in the Connection Properties window

See the chapter describing working with remote targets in RealView Debugger v1.7 Target Configuration Guide for details on how to make connections using these settings.

Using Trace

Developers working on Sun Solaris or Red Hat Linux can use Trace with RVISS.

By default, RealView Debugger is automatically configured with tracing enabled for ARM targets using preset values stored in the Logic_Analyzer settings group in the Advanced_Information block. These settings are not used with RVISS.

--- Note ---

RealView Debugger v1.7 enables you to access Trace and profiling features without having to purchase a separate license. These features are part of the core product.

See the chapter describing tracing in RealView Debugger v1.7 Extensions User Guide for full details on using this extension.

Using the DSP

The DSP support in RealView Debugger is invoked by connecting the debugger to an Oak or TeakLite processor. Developers working on Sun Solaris or Red Hat Linux can connect to a simulated target on a remote workstation or connect to target hardware using Multi-ICE direct connect. See the chapter describing working with remote targets in RealView Debugger v1.7 Target Configuration Guide for details.
Note

RealView Debugger DSP support is a separately licensed component. See the chapter describing DSP support in RealView Debugger v1.7 Extensions User Guide for full details on using this extension.
B.4 Changes to GUI and general user information

This section describes:
- Getting started with RealView Debugger
- Column resizing on page B-9
- Saving favorites on page B-10
- Changes to the desktop on page B-11
- RealView Debugger examples on page B-12.

B.4.1 Getting started with RealView Debugger

The RealView Debugger Control Panel, RVDEBUGCP, provides access to the debugger for developers using Sun Solaris and Red Hat Linux. Start the debugger to see this window, shown in Figure B-2.

![RealView Debugger Control Panel](image)

Figure B-2 RealView Debugger Control Panel

By default, launching the Control Panel starts RealView Debugger without a splash screen.

Click on the required icon to launch:

**rvdebug**  
Access startup options for RealView Debugger.  
Double-click on this icon to start the debugger in default mode and display the Code window.

**Editor**  
Access startup options for mwedit, the standalone editor installed as part of RealView Debugger. Use this to edit source files and work on project files.  
Double-click on this icon to start mwedit in default mode and display the standalone editor window.
Running RealView Debugger from the Control Panel

Left-click on the `rvdebug` icon to specify arguments when using the command-line method of starting RealView Debugger:

**Start With Arguments...**

Displays a dialog box where you can specify startup options, for example to specify a workspace to use in this session or to write to a log file.

**Set Default Arguments...**

Displays the Tools Attributes dialog box where you can specify the start directory and arguments for running RealView Debugger. If you set these, they are used each time the debugger runs from the Control Panel.

See Chapter 1 *Starting to use RealView Debugger* for details of startup options and command-line arguments.

--- **Note** ---

If you are using Sun Solaris or Red Hat Linux, you can use the `-cmd` argument to start the command-line debugger, for example `rvdebug -cmd`.

Running an editor from the Control Panel

Left-click on the `Editor` icon to specify arguments when starting mwedit:

**Start With Arguments...**

Displays a dialog box where you can specify startup options, for example to start the editor with a file loaded.

**Set Default Arguments...**

Displays the Tools Attributes dialog box where you can specify the start directory and arguments for running mwedit. If you set these, they are used each time the standalone editor runs from the Control Panel.

--- **Note** ---

If you are using mwedit in standalone mode, certain menu options are not available. For example, you cannot access workspace options, connection details, or project-related operations from the mwedit menu bar. These options are available if you start the standalone editor from within the Code window.

--- See *RealView Debugger v1.7 Project Management User Guide* for details on using editors in RealView Debugger. ---
B.4.2 Column resizing

When using the Code window, the following panes use two columns to display debug data:

- Call Stack
- Break/Tracepoints
- Watch
- Process Control.

To make the display easier to read, you can change the size of the first column using a new menu option available only on Sun Solaris and Red Hat Linux:

1. Select **File → Reload Image to Target** to reload the image dhrystone.axf.
2. Click on the **Src** tab to view the source file dhry_1.c.
3. Set a simple breakpoint by double-clicking on line 150.
4. Click **Go** to start execution.
5. Enter 5000 when asked for the number of runs.
   The program starts and then stops when execution reaches the breakpoint at line 150. The red box marks the location of the **Program Counter (PC)** when execution stops.
6. Select **View → Pane Views → Call Stack** to view the Call Stack pane.
7. Right-click somewhere in the header columns to display the context menu, shown in Figure B-3.

![Figure B-3 Column resizing in the Watch pane](image)

8. Select the option **Set Width Of Column 1** to display the prompt box where you can specify the size you want, shown in Figure B-4 on page B-10.
Figure B-4 Column resizing prompt box

When the prompt box first appears, it contains the current setting. Enter a value between 1 and 128.

9. Click **Set** to confirm the new setting. Click **Cancel** to leave the size unchanged.

If you change the size of a column, it holds for all windows in the current session. Default sizes are restored at each start up.

You cannot change the size of columns in the following panes:
- Register
- Stack
- Symbol Browser
- Memory.

If you resize columns be aware of the following:
- You can only resize the first column in any two-column display.
- Columns can only be adjusted to within one monospaced character position.
- The second column is automatically set to accommodate the longest item.
- Changing the column size applies to all tabs in a multitab display, that is the Watch, Call Stack, and Process Control panes.

### B.4.3 Saving favorites

Be aware of the following when working on Sun Solaris or Red Hat Linux:
- The history file, `exphist.sav`, is only created if you create and save a favorite, for example a breakpoint or watchpoint.
- The history file is not controlled in the same way when you access an **Open...** dialog.

When the file has been created and saved, it is updated at the end of each debugging session in the usual way.
### B.4.4 Changes to the desktop

This section summarizes differences when using the RealView Debugger desktop on Sun Solaris or Red Hat Linux.

#### Code window

Be aware of the following when working in the default Code window:

- Code windows are identified by a title bar. This changes as you open new windows, for example RVDEBUG, RVDEBUG_1, RVDEBUG_2.
- The Code window title bar identifies the vehicle you are using to make the connection and the connection number. The target processor is not shown.
- Code windows do not have a Color Box to identify target connections.
- Other windows, such as the Resource Viewer, do not include a Color Box to identify the calling Code window.
- Code window panes cannot float.
- Tooltips are not available if you hover over a toolbar button.
- The editing control called Tooltip Evaluation that provides hover-style evaluation in different code views is not available.
- The Code window main menu includes a Help menu. This is located at the right of the default window.
- Select Help → About... from the RealView Debugger main menu to display the About Box Information dialog where you can submit a Software Problem Report.
- There are no status display areas at the bottom of the Code window.
- Developers working on Red Hat Linux must use the keyboard shortcut Ctrl+Shift+F10 to step over functions (see Using shortcuts on page 3-9).

#### Pane controls

Panes do not include title bars to describe their content. However, each pane contains the controls:

**Pane Content**

Click this button to display the Pane Content menu where you can change the debug view in the pane.

The selected option in the menu indicates the current view.
The visual controls are at the bottom of the Pane Content menu. Use these to hide the pane. The option to float a pane is not available.

**Pane Menu**

Click this button to display the Pane menu.

Use this to:

- change the display format
- change how pane contents are updated
- extract data from the pane.

The options available from this menu depend on the pane.

**Expand/Collapse Pane**

Use the pane slide controls to change the size of a pane.

You cannot switch the Side pane from its location to the right of the File Editor pane.

### B.4.5 RealView Debugger examples

Be aware of the following when working on Sun Solaris or Red Hat Linux.

Depending on your installation, the RealView Debugger examples for developing applications on ARM targets, are installed in the examples directory in:

`install_directory/RVDS/Examples/...`

The dhrystone example project, and the executable built by this project, are used in the documentation suite.

**Note**

The `-D` compiler switch is set to `TIME` to control how timing measurements are made on Sun Solaris and Red Hat Linux platforms.

See the tutorial in *RealView Debugger v1.7 Essentials Guide* for more details on building this example project.

This location is also used, by default, if you choose a Custom installation and specify that the RealView Debugger examples are installed.

**Tools support examples**

If you choose a Custom installation to install RealView Debugger Tools support for Oak-Teaklite, DSP examples are installed in the examples directory:

`install_directory/RVD/Tools/Oak-Teaklite/.../demo_DSPG`
The items in this glossary are listed in alphabetical order, with any symbols and numerics appearing at the end.

**Access-provider connection**
A debug target connection item that can connect to one or more target processors. The term is normally used when describing the RealView Debugger Connection Control window.

**Address breakpoint**
A type of breakpoint.

*See also* Breakpoint.

**ADS**
See ARM Developer Suite.

**Angel**
Angel is a software debug monitor that runs on the target and enables you to debug applications running on ARM-based hardware. Angel is commonly used where a JTAG emulator is not available.

**ARM Developer Suite (ADS)**
A suite of software development applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of RISC processors. ADS is superseded by RealView Developer Suite (RVDS).

*See also* RealView Developer Suite.
Glossary

**ARM state**
A processor that is executing ARM (32-bit) instructions is operating in ARM state.  
*See also* Thumb state

**Asynchronous execution**
*Asynchronous execution* of a command means that the debugger accepts new commands as soon as this command has been started, enabling you to continue do other work with the debugger.

**Backtracing**
*See* Call Stack.

**Big-endian**
Memory organization where the least significant byte of a word is at the highest address and the most significant byte is at the lowest address in the word.  
*See also* Little-endian.

**Board**
RealView Debugger uses the term *board* to refer to a target processor, memory, peripherals, and debugger connection method.

**Board file**
The *board file* is the top-level configuration file, normally called `rvdebug.brd`, that references one or more other files.

**Breakpoint**
A user defined point at which execution stops in order that a debugger can examine the state of memory and registers.  
*See also* Hardware breakpoint and Software breakpoint.

**Call Stack**
This is a list of procedure or function call instances on the current program stack. It might also include information about call parameters and local variables for each instance.

**Conditional breakpoint**
A breakpoint that halts execution when a particular condition becomes true. The condition normally references the values of program variables that are in scope at the breakpoint location.

**Context menu**
*See* Pop-up menu.

**Core module**
In the context of Integrator, an add-on development board that contains an ARM processor and local memory. Core modules can run stand-alone, or can be stacked onto Integrator motherboards.  
*See also* Integrator.

**Current Program Status Register (CPSR)**
*See* Program Status Register.

**DCC**
*See* Debug Communications Channel.
Debug Agent (DA)  The Debug Agent resides on the target to provide target-side support for Running System Debug (RSD). The Debug Agent can be a thread or built into the RTOS. The Debug Agent and RealView Debugger communicate with each other using the debug communications channel (DCC). This enables data to be passed between the debugger and the target using the ICE interface, without stopping the program or entering debug state.

See also Running System Debug.

Debug Communications Channel (DCC)  A debug communications channel enables data to be passed between RealView Debugger and the EmbeddedICE logic on the target using the JTAG interface, without stopping the program flow or entering debug state.

Debug With Arbitrary Record Format (DWARF)  ARM code generation tools generate debug information in DWARF2 format.

Deprecated  A deprecated option or feature is one that you are strongly discouraged from using. Deprecated options and features will not be supported in future versions of the product.

Digital Signal Processor (DSP)  DSPs are special processors designed to execute repetitive, maths-intensive algorithms. Embedded applications might use both ARM processor cores and DSPs.

Doubleword  A 64-bit unit of information.

DSP  See Digital Signal Processor.

DWARF  See Debug With Arbitrary Record Format.

ELF  Executable and Linking Format. ARM code generation tools produce objects and executable images in ELF format.

Embedded Trace Macrocell (ETM)  A block of logic, embedded in the hardware, that is connected to the address, data, and status signals of the processor. It broadcasts branch addresses, and data and status information in a compressed protocol through the trace port. It contains the resources used to trigger and filter the trace output.

EmbeddedICE logic  The EmbeddedICE logic is an on-chip logic block that provides TAP-based debug support for ARM processor cores. It is accessed through the TAP controller on the ARM core using the JTAG interface.

See also IEEE1149.1.
Emulator

In the context of target connection hardware, an emulator provides an interface to the pins of a real core (emulating the pins to the external world) and enables you to control or manipulate signals on those pins.

Endpoint connection

A debug target processor, normally accessed through an access-provider connection.

ETM

See Embedded Trace Macrocell.

ETV

See Extended Target Visibility.

Extended Target Visibility (ETV)

Extended Target Visibility enables RealView Debugger to access features of the underlying target, such as chip-level details provided by the hardware manufacturer or SoC designer.

Floating Point Emulator (FPE)

Software that emulates the action of a hardware unit dedicated to performing arithmetic operations on floating-point values.

FPE

See Floating Point Emulator.

Halfword

A 16-bit unit of information.

Halted System Debug (HSD)

Usually used for RTOS aware debugging, Halted System Debug (HSD) means that you can only debug a target when it is not running. This means that you must stop your debug target before carrying out any analysis of your system. With the target stopped, the debugger presents RTOS information to the user by reading and interpreting target memory.

See also Running System Debug.

Hardware breakpoint

A breakpoint that is implemented using non-intrusive additional hardware. Hardware breakpoints are the only method of halting execution when the location is in Read Only Memory (ROM). Using a hardware breakpoint often results in the processor halting completely. This is usually undesirable for a real-time system.

See also Breakpoint and Software breakpoint.

HSD

See Halted System Debug.

IEEE 1149.1

The IEEE Standard that defines TAP. Commonly (but incorrectly) referred to as JTAG.

See also Test Access Port

Integrator

A range of ARM hardware development platforms. Core modules are available that contain the processor and local memory.
Joint Test Action Group (JTAG)
An IEEE group focussed on silicon chip testing methods. Many debug and programming tools use a Joint Test Action Group (JTAG) interface port to communicate with processors. For further information refer to IEEE Standard, Test Access Port and Boundary-Scan Architecture specification 1149.1 (JTAG).

JTAG
See Joint Test Action Group.

JTAG interface unit
A protocol converter that converts low-level commands from RealView Debugger into JTAG signals to the processor, for example to the EmbeddedICE logic and the ETM.

Little-endian
Memory organization where the least significant byte of a word is at the lowest address and the most significant byte is at the highest address of the word.

See also Big-endian.

Multi-ICE
A JTAG-based tool for debugging embedded systems.

Pop-up menu
Also known as Context menu. A menu that is displayed temporarily, offering items relevant to your current situation. Obtainable in most RealView Debugger windows or panes by right-clicking with the mouse pointer inside the window. In some windows the pop-up menu can vary according to the line the mouse pointer is on and the tabbed page that is currently selected.

Processor core
The part of a microprocessor that reads instructions from memory and executes them, including the instruction fetch unit, arithmetic and logic unit and the register bank. It excludes optional coprocessors, caches, and the memory management unit.

Profiling
Accumulation of statistics during execution of a program being debugged, to measure performance or to determine critical areas of code.

Program Status Register (PSR)
Contains information about the current execution context. It is also referred to as the Current PSR (CPSR), to emphasize the distinction between it and the Saved PSR (SPSR), which records information about an alternate processor mode.

PSR
See Program Status Register.

RDI
See Remote Debug Interface.

RealView ARMulator ISS (RVISS)
The most recent version of the ARM simulator, RealView ARMulator ISS is supplied with RealView Developer Suite. It communicates with a debug target using RV-msg, through the RealView Connection Broker interface, and RDI.

See also RDI and RealView Connection Broker.
**RealView Compilation Tools (RVCT)**

RealView Compilation Tools is a suite of tools, together with supporting documentation and examples, that enables you to write and build applications for the ARM family of RISC processors.

**RealView Connection Broker**

RealView Connection Broker is an execution vehicle that enables you to connect to simulator targets on your local system, or on a remote system. It also enables you to make multiple connections to the simulator.

*See also* RealView ARMulator ISS.

**RealView Debugger Trace**

Part of the RealView Debugger product that extends the debugging capability with the addition of real-time program and data tracing. It is available from the Code window.

**RealView ICE (RVI)**

A JTAG-based debug solution to debug software running on ARM processors.

**Remote Debug Interface (RDI)**

The *Remote Debug Interface* (RDI) is an ARM standard procedural interface between a debugger and the debug agent. RDI gives the debugger a uniform way to communicate with:

- a simulator running on the host (for example, RVISS)
- a debug monitor running on hardware that is based on an ARM core accessed through a communication link (for example, Angel)
- a debug agent controlling an ARM processor through hardware debug support (for example, RealView ICE or Multi-ICE).

**Remote_A**

Remote_A is a software protocol converter and configuration interface. It converts between the RDI 1.5 software interface of a debugger and the Angel Debug Protocol used by Angel targets. It can communicate over a serial or Ethernet interface.

**RSD**

*See* Running System Debug.

**RTOS**

Real Time Operating System.

**Running System Debug (RSD)**

Used for RTOS aware debugging, *Running System Debug* (RSD) means that you can debug a target when it is running. This means that you do not have to stop your debug target before carrying out any analysis of your system. RSD gives access to the application using a *Debug Agent* (DA) that resides on the target. The Debug Agent is scheduled along with other tasks in the system.

*See also* Debug Agent and Halted System Debug.

**RVCT**

*See* RealView Compilation Tools.

**RVISS**

*See* RealView ARMulator ISS.
**Scan chain**
A scan chain is made up of serially-connected devices that implement boundary-scan technology using a standard JTAG TAP interface. Each device contains at least one TAP controller containing shift registers that form the chain. Processors might contain several shift registers to enable you to access selected parts of the device.

**Scope**
The range within which it is valid to access such items as a variable or a function.

**Script**
A file specifying a sequence of debugger commands that you can submit to the command-line interface using the `include` command.

**Semihosting**
A mechanism whereby I/O requests made in the application code are communicated to the host system, rather than being executed on the target.

**Simulator**
A simulator executes non-native instructions in software (simulating a core).

**Software breakpoint**
A breakpoint that is implemented by replacing an instruction in memory with one that causes the processor to take exceptional action. Because instruction memory must be altered software breakpoints cannot be used where instructions are stored in read-only memory. Using software breakpoints can enable interrupt processing to continue during the breakpoint, making them more suitable for use in real-time systems.

See also Breakpoint and Hardware breakpoint.

**Software Interrupt (SWI)**
An instruction that causes the processor to call a programmer-specified subroutine. Used by the ARM standard C library to handle semihosting.

**SPSR**
Saved Program Status Register.

See also Program Status Register.

**SWI**
See Software Interrupt.

**Synchronous execution**
*Synchronous execution* of a command means that the debugger stops accepting new commands until this command is complete.

**Synchronous starting**
Setting several processors to a particular program location and state, and starting them together.

**Synchronous stopping**
Stopping several processors in such a way that they stop executing at the same instant.

**TAP**
See Test Access Port.
TAP Controller
Logic on a device which enables access to some or all of that device for test purposes. The circuit functionality is defined in IEEE1149.1.

See also Test Access Port and IEEE1149.1.

Target
The target hardware, including processor, memory, and peripherals, real or simulated, on which the target application is running.

Target vehicle
Target vehicles provide RealView Debugger with a standard interface to disparate targets so that the debugger can connect easily to new target types without having to make changes to the debugger core software.

Target Vehicle Server (TVS)
Essentially the debugger itself, this contains the basic debugging functionality. TVS contains the run control, base multitasking support, much of the command handling, and target knowledge, such as memory mapping, lists, rule processing, board file and .bcd files, and data structures to track the target environment.

Test Access Port (TAP)
The port used to access the TAP Controller for a given device. Comprises TCK, TMS, TDI, TDO, and nTRST (optional).

Thumb state
A processor that is executing Thumb (16-bit) instructions is operating in Thumb state.

See also ARM state

Tracepoint
A tracepoint can be a line of source code, a line of assembly code, or a memory address. In RealView Debugger, you can set a variety of tracepoints to determine exactly what program information is traced.

Tracing
The real-time recording of processor activity (including instructions and data accesses) that occurs during program execution. Trace information can be stored either in a trace buffer of a processor, or in an external trace hardware unit. Captured trace information is returned to the Analysis window in RealView Debugger where it can be analyzed to help identify a defect in program code.

Trigger
In the context of breakpoints, a trigger is the action of noticing that the breakpoint has been reached by the target and that any associated conditions are met.

In the context of tracing, a trigger is an event that instructs the debugger to stop collecting trace and display the trace information around the trigger position, without halting the processor. The exact information that is displayed depends on the position of the trigger within the buffer.

TVS
See Target Vehicle Server.
**Vector Floating Point (VFP)**
A standard for floating-point coprocessors where several data values can be processed by a single instruction.

**VFP**
*See Vector Floating Point.*

**Watch**
A watch is a variable or expression that you require the debugger to display at every step or breakpoint so that you can see how its value changes. The Watch pane is part of the RealView Debugger Code window that displays the watches you have defined.

**Watchpoint**
In RealView Debugger, this is a hardware breakpoint.

**Word**
A 32-bit unit of information.
Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.

A

About this book viii
Add/Copy/Edit Memory Map
  dialog 5-5
Add/Edit Macros
  dialog 9-8
Audience, intended viii
Auto-projects
  changing settings 2-14
  closing 2-15
  creating for images 2-12
  displaying settings 2-13
  in the Process Control pane 2-11, 2-12
  properties 2-12, 2-13
  saving settings 2-15
  see also Projects
Autoscope 3-3
  in assembly language 3-3
  red box 3-3

B

Banks
  of registers 6-8, 6-9
Board file 2-2
Book, about this viii
Breakpoint units 4-10
  in Break/Tracepoints pane 4-10
Breakpoints 4-2
  Access 4-39
  actions 4-22
  and memory mapping 4-3
  and stepping 4-3
  attaching macros 4-26
  breakpoint types 4-2
  breakpoint units 4-10
  chaining 4-5
  class 4-15
  clearing 4-43, 4-44
  clearing all 4-20, 4-44
  clearing quickly 4-11
  complex 4-2, 4-34
  Complex Breakpoints menu 4-34
  conditional breakpoints 4-17, 4-22
  copying in the Break/Tracepoints pane 4-41
  creating a favorite 4-45
  Debug menu mappings 4-42
  deleting 4-44
  deleting all 4-44
  disable at cursor 4-20
  disabling 4-43
  editing in the Break/Tracepoints pane 4-41
  enable at cursor 4-20
  EXEC 4-39
  hardware 4-4, 4-8, 4-28
  hardware limitations 4-5
  hardware support in a DSP 4-31, 4-33
  in the Break/Tracepoints pane 4-37, 4-39
  INSTR 4-39
  managing actions 4-24
  managing qualifiers 4-23
  markers 4-10, 4-39
mode tests 4-5
named 4-18
operations 4-20
pass counts 4-5, 4-18, 4-23, 4-33
Processor Events 4-19
qualifiers 4-22
Read 4-39
red disc 4-7, 4-10
removing 4-44
removing all 4-44
saving as favorites 4-45, 4-46
saving as history 4-20
setting hardware breakpoints 4-28
setting hardware breakpoints using a DSP 4-33
setting hardware tests 4-30
setting in disassembly-level view 4-8
setting in other ways 4-11
setting in source-level view 4-7
setting in the Memory pane 4-11
setting in the Stack pane 4-11
setting on a branch 4-9
setting on a history 4-20
setting on a symbol 4-8
setting on an instruction 4-9
setting on inline functions 4-10
setting on multiple statements 4-10
setting on the statement 4-7
setting simple breakpoints 4-17
setting using drag-and-drop 4-11
simple 4-2, 4-13
Simple Breakpoints menu 4-13
size tests 4-5
software 4-4, 4-8
toggle at cursor 4-20
using a mask 4-31
using a range 4-31
using with macros 9-6
viewing actions 4-25
viewing in disassembly-level view 4-10
viewing in source-level view 4-10
viewing qualifiers 4-25
watchpoints 4-5
white disc 4-10
Write 4-39
yellow disc 4-10
Browsers
accessing 8-2, 8-18
applying a filter 8-15
browsing C++ classes 8-3, 8-16
browsing functions 8-7
browsing modules and files 8-4
browsing variables 8-11
filter metacharacters 8-15
Function List 4-12, 8-2, 8-7
in RealView debugger 8-2
Module/File List 8-2, 8-4
Register List 8-2
scope 8-2
using filters 8-14
Variable List 8-2, 8-11, 8-16
-C
Call Stack 6-32
see also Stack
Cancel button
in the Code window 3-2
Channel viewers 6-12
Code patching
assembly patching 7-3
Code views 2-9, 2-21, 3-3
Dsm tab 2-9, 2-21, 3-3
setting disassembly mode 3-5
source code tab 2-9, 2-21, 3-3
Src tab 2-9, 2-21, 3-3
toggling 3-13
Colors
in memory map 5-8
in Memory pane 6-20
in Register pane 6-4
Command line
arguments for RealView Connection Broker 1-6
arguments for RealView Debugger 1-2
starting RealView Connection Broker 1-6
starting RealView Network Broker 1-6
Command line arguments
-aws 1-3
-bat 1-2
-cmd 1-2
-exec 1-3
-Home 1-3
-inc 1-3
-install 1-2, 1-7
-jou 1-3
-log 1-3
-nologo 1-4
-s 1-3
-user 1-3, 1-7
Commands
from the debuggee 3-14
in a journal file 3-14
logging 3-14
logging from the debuggee 3-14
logging from the debugger 3-14
pending 3-2
purging 3-2
queue 3-2
queue rules 3-2
to the debugger 3-14
Comments from users xiii
Complex breakpoints
see Breakpoints
Conditional breakpoints
see Breakpoints
Connecting 2-2
Connection Broker 1-6
local host connection 1-6
remote host connection 1-6
shut down in remote mode 1-7
Context
controls 6-30
see Scope
Customizing
registers 6-14
Cycles
in Register pane 6-10
RealView ARMulator ISS 6-10
C++
browsing classes 8-16
classes in the Symbol Browser 8-16
finding a class definition 8-17
viewing classes 8-17
D
DA
Debug Agent 4-4
Data values
saving as favorites 6-44
DCC 6-12
DCC semihosting 6-12
enabling 6-13
Debug Agent 4-4
Debug communications channel 6-12
Debug menu 3-10
Debug target
configuration 2-2
connecting 2-2
Connection Broker 1-6
memory mapping 5-2, 5-7, 5-8
resetting processor 2-21, 3-12
Debugger
defining in workspace 10-14
internal variables 6-10
internals 6-10, 6-11
internals in RealView ARMulator
ISS 6-10
statistics 6-10
Dialog box
Add/Copy/Edit Memory Map 5-5
Add/Edit Macros 9-8
Fill Memory with Pattern 7-19
Find in Files 8-17
Flash Memory Control 7-10, 7-11
Function List 4-12, 8-7
HW Break if in Range 4-34
HW Break if X, then if Y 4-35
HW Break on Data Value match 4-36
HW While in func/range, Break if X 4-35
Interactive Memory Setting 7-13
Interactive Register Setting 7-16
Load File to Target 2-3
Module/File List 8-4
New/Edit Favorite 4-45
Select File to Include Commands from 3-16
Select File to Journal to 3-15
Select File to log the STDIO to 3-15
Select File to Log to 3-15
Set Address/Data Break/Tracepoint 4-13
Simple Break if X 4-17
Simple Break if X, N times 4-17
Simple Break if X, when Y is True 4-18
Upload/Download file from/to Memory 7-16
Variable List 8-11
Directories
examples 1-9
Flash examples 1-9
home 1-8
home and the user ID 1-8
home and the Windows login ID 1-8
installation 1-8
used in RealView Debugger 1-8
E
Environment variables
JOULOG_UNBUF 3-14
RVDEBUG_HOME 1-5
RVDEBUG_INSTALL 1-5
RVDEBUG_SHARE 1-5
EXEC
hardware breakpoints 4-39
in Break/Tracepoints pane 4-39
Execution controls 3-7
from the Debug menu 3-11
submitting commands 3-2
using shortcuts 3-9
Expression pointer 6-30
F
Favorites
breakpoint 4-20, 4-45
data value 6-44
watch 6-42
Feedback xiii
Files
board file 2-2
including from the Debug menu 3-16
journal 3-14
log 3-14
STDIlog 3-14
*.apr 2-11
*.aws 10-2, 10-9
*.brd 2-2
*.ini 10-2, 10-8, 10-11
*.prj 2-11
*.sav 4-20, 4-45, 4-46, 6-42, 6-44
Fill Memory with Pattern
dialog 7-19
Find in Files
C++ Class definition 8-17
dialog 8-17
Flash
definition files 7-6
directories 1-9
examples 1-9
FME files 7-7
setting interactively 7-6
Flash Memory Control
dialog 7-10, 7-11
Flash MEthod files
see Flash
Force scope 3-4
Frame pointer 6-27, 6-30
Function List
dialog 4-12, 8-7
G
Global options
in workspace 10-2, 10-11
Glossary Glossary-1
Go button
in the Code window 3-7
Go to Cursor button
in the Code window 3-9
Go until Return button
in the Code window 3-9
H
Halted System Debug
HSD 4-4
Hardware breakpoints
see Breakpoints
Heap 5-12
see Memory
High-level Step button
in the Code window 3-7
High-level Step Into button
in the Code window 3-8
Home
see Directories
Index

HSD
- Halted System Debug 4-4
- switching to RSD 4-4

HW Break if in Range
dialog 4-34
HW Break if X, then if Y
dialog 4-35
HW Break on Data Value match
dialog 4-36
HW While in func/range, Break if X
dialog 4-35

Images
- auto-set PC on load 2-4
- connecting on loading 2-6
- debugging multiple images 2-19
- disconnecting whilst loaded 2-22
- load settings in project 2-14
- loading 2-2, 5-5
- loading above 0x8000 2-20
- loading from a user-defined project 2-3
- loading from the command line 2-6
- loading from the Load File to Target dialog box 2-3
- loading from the Process Control pane 2-5
- loading multiple 2-19
- loading progress 2-7
- managing 2-8
- passing arguments on loading 2-4, 2-7
- passing arguments on reloading 2-23
- passing target name on loading 2-4
- Processor State 2-7
- quick loading 2-5
- reloading 2-21, 2-22
- removing all details 2-22
- runtime indicator 2-7
- set PC to entry point on load 2-5
- setting PC manually 2-20
- setting PC on load 2-4
- symbol data 2-18
- unloading 2-21
- unloading and auto-projects 2-22

viewing in the Code window 2-8
viewing in the Process Control pane 2-10
Include files 3-14
from the Debug menu 3-16
Journal files 3-14
Log files 3-14
STDIOlog files 3-14
INSTR
- in Break/Tracepoints pane 4-39
- software breakpoints 4-39
Intended audience viii
Interactive Memory Setting
dialog 7-13
Interactive Register Setting
dialog 7-16

Log files
- at start-up 3-15, 3-17
- closing 3-16
- in the Status line 3-15
- output buffering 3-14
- viewing 3-16
Low-level Step Into button
in the Code window 3-8
Low-level Step Over button
in the Code window 3-8

Macros 9-2
- attaching 9-6
- calling 9-5, 9-14
- calling from the Debug menu 9-14
- copying 9-13
- creating 9-8
- declaring variables 9-9
- defining 9-5
- deleting 9-13
- editing 9-11
- in symbol table 9-2, 9-5, 9-9, 9-14
- in the symbol table 2-18
- including a definition file 9-5, 9-14
- loading from a project 9-15
- loading on connection 9-14
- precedence 9-6
- properties 9-3
- return values 9-6
- stopping 9-7
- using debugger commands 9-3, 9-9
- using with breakpoints 4-26, 9-6
- viewing 9-10

Mapping
- source files 3-20

Margin
- breakpoint markers 4-10
- red disc 4-10
- setting breakpoints 4-7
- white disc 4-10
- yellow disc 4-10

Memory
- and breakpoints 4-3
- changing contents 6-20, 6-23
- data formats 6-19
- disabling memory mapping 5-4
Index

display colors   6-20
downloading to a file   7-16
editing map entries   5-5
enabling memory mapping   5-4
filling with a pattern   7-19
formatting options   6-16
generating linker command files   5-13
in RealView ARMulator ISS   5-12
interactive operations   7-2
interactive operations examples   7-13
interactive operations from the Debug menu   7-3
interactive operations from the Memory pane   7-5
interactive operations from the Stack pane   7-5
map configuration   5-8
mapping   5-2
mapping and breakpoints   4-3
setting access size   5-6, 6-23
setting interactively   7-13
setting stack and heap   5-12
setting top of memory   5-12
specifying range   7-17
verifying against a file   7-18
viewing contents   6-15
viewing for multiple targets   6-25
viewing the map   5-7
working with Flash   7-6

Memory map
display colors   5-8
editing   5-5
setting up   5-5
update based on processor registers   5-11

Menus
Break/Tracepoints   4-40
Complex Breakpoints   4-34
C++ Class   8-17
C++ Function   8-17
Debug   3-10
Disassembly View   3-5
Execution Control   3-11, 3-12
File   2-3
Find   8-2
Memory/Register Operations   7-3
New Actions   4-24
New Qualifiers   4-23
Pane   4-42
Settings options   10-4
Simple Breakpoints   4-13
Tools   10-8
View   2-5
Workspace   10-3

Metacharacters
in browser filters   8-15
using in browsers   8-15

Mode tests
used in hardware breakpoints   4-5

Module/File List
dialog   8-4

Multi-ICE
DCC   6-12
DCC semihosting   6-12, 6-13
debugger internals   6-11
Standard semihosting   6-12

N
Named breakpoint
see Breakpoints

Network Broker   1-6
New/Edit Favorite
dialog   4-45

Numbers
turn on lines   2-9, 3-4, 3-7, 4-7, 6-2, 7-13, 9-8

O
Objects
closing open   10-5

Options window   10-8

P
Panes
Break/Tracepoints   4-10, 4-37, 4-38
Call Stack   6-32
File Editor   3-3
Memory   6-15, 6-29
Process Control   2-10
Register   6-2
Stack   6-27
Watch   6-36
Pass counts
used in hardware breakpoints   4-5, 4-18, 4-33
used in software breakpoints   4-23
PC   2-4
see also Program Counter

Pending
commands   3-2

Persistence info
breakpoints history   4-20
favorites   4-45, 6-42

Process
in Process Control pane   2-10, 2-20
multiple images   2-19

Process Control
properties   2-12
tabs   2-10
Process Control pane
type ahead   2-11

Processor
reset   2-21
resetting   2-21

Processor Events
see Breakpoints

Processor mode
in register view   6-8
privileged mode   6-8

Program Counter
locating   3-4
red box   3-3, 3-4
reset to entry point   3-5
see also Scope
setting on load   2-4

Projects
associated images   2-16
auto-projects   2-12, 2-13
closing   2-16
defining in workspace   10-6
Project Properties window   3-19
searching source files   3-17
source files mapping   3-20

Q
Queue
command handling   3-2

Index-5
Index

commands 3-2

R
RealView ARMulator ISS
and the RESET command 4-44
core models 6-4
cycle counting 6-10
debugger internals 6-10
hardware breakpoints 4-5
in examples viii
loading images 2-2
stopping during semihosting input
3-9
top of memory 5-12
top of stack 5-3
RealView Connection Broker
arguments 1-6
starting from the command line 1-6
RealView Debugger
arguments 1-2
connecting to a target 2-2
debugger internals 6-10, 6-11
directories 1-8
environment variables 1-5
image loading 5-5
memory mapping 5-2
starting 1-2
target configuration 2-2
using macros 9-2
using Red Hat Linux B-2
using Sun Solaris B-2
RealView ICE
DCC 6-12
DCC semihosting 6-12, 6-13
debugger internals 6-11
Standard semihosting 6-12
RealView Network Broker
starting from the command line 1-6
Red Hat Linux
changes to the Code window B-11
changes to the desktop B-11
changes to the Pane controls B-11
Code window B-11
column resizing B-9
configuration files for RealView ARMulator ISS B-4
collection of commands 3-2
configuration files for Simulator Broker B-4
creating the history file B-10
exphist.sav B-10
following examples B-12
Front Panel B-7
installing examples B-12
pane controls B-11
saving favorites B-10
target configuration B-4
using DSP B-5
using Trace B-5
Registers
ARM processors 6-8
banked out 6-8, 6-9
banks 6-8, 6-9
changing contents 6-5
cycle counting 6-10
DCC semihosting 6-13
defining custom registers 6-14
different views 6-6
display colors 6-4
formatting options 6-4
interactive operations 7-2
interactive operations examples 7-13
interactive operations from the Debug menu 7-3
processor mode 6-8
semihosting 6-13
setting interactively 7-15
updating contents 6-6
viewing 6-8
viewing contents 6-2
viewing debugger internals 6-10, 6-11
viewing for multiple targets 6-8
Reloading
images 2-22
using arguments 2-23
RSD
Running System Debug 4-4
switching to HSD 4-4
RTOS
Debug Agent 4-4
HSD 4-4
RSD 4-4
Running System Debug 4-4
Run-time stack
see Stack
RVDEBUG_HOME 1-5
RVDEBUG_INSTALL 1-5
RVDEBUG_SHARE 1-5
Scope
autoconfiguring 3-18
configuring 3-19
search rules 3-17, 3-18
Searching
source files 3-17, 3-20
source search rules 3-17
Scope controls 6-30, 6-35
Set Address/Data Break/Tracepoint
dialog 4-13
Settings
in Workspace A-1
save on exit 10-4
Settings options 10-4
Simple Break if X
dialog 4-17
Simple Break if X, N times
dialog 4-17
Simple Break if X, when Y is True
dialog 4-18
Simple breakpoints
see Breakpoints
Software breakpoints
see Breakpoints
Solaris
see Sun Solaris
Source search prompt 3-18
Stack
breakpoints 6-31
changing contents 6-29
context controls 6-30, 6-35
data formats 6-29
EP 6-30
formatting options 6-28
FP 6-27, 6-30
in RealView ARMulator ISS 5-3
in the Stack pane 6-26, 6-27
pointer 6-26
see also Memory
SP 6-27, 6-30
Stack down button
in the Code window 6-30
Stack pointer 6-27, 6-30
default 6-30
Stack up button
in the Code window 6-30
Starting
RealView Debugger 1-2
STDIOlog files
at start-up 3-15, 3-17
closing 3-16
in the Status line 3-15
output buffering 3-14
viewing 3-16
Stepping
and breakpoints 4-3
Stop Execution button
in the Code window 3-9
Structure of this book ix
Sun Solaris
changes to the Code window B-11
changes to the desktop B-11
changes to the Pane controls B-11
Code window B-11
column resizing B-9
configuration files for RealView
ARMulator ISS B-4
configuration files for Simulator
Broker B-4
creating the history file B-10
expolist.sav B-10
following examples B-12
Front Panel B-7
installing examples B-12
pane controls B-11
saving favorites B-10
target configuration B-4
using DSP B-5
using Trace B-5
Symbols
loading 2-4, 2-18
macros in the symbol table 2-18
symbol table 2-18

T
Target
see Debug target
Terminology Glossary-1
Threads
HSD 4-4
in the Break/Tracepoints pane 4-38, 4-41
markers 4-11
RSD 4-4
Toolbars
customizing 10-11
Tracepoints 3-11
class 4-15
clearing 4-44
clearing all 4-20, 4-44
deleting 4-44
deleting all 4-44
disabling 4-43
in the Break/Tracepoints pane 4-37, 4-38, 4-41
in the Set Address/Data
Break/Tracepoints dialog 4-13
markers 4-11
removing 4-44
removing all 4-44
type 4-15

U
UNIX
see Sun Solaris or Red Hat Linux
Unloading
images 2-21
Upload/Download file from/to Memory
dialog 7-16
User ID
creating the home directory 1-8
User’s comments xiii

V
Variable List
dialog 8-11

W
Watches
creating a favorite 6-42
editing 6-42
in the Watch pane 6-36
managing 6-39
saving as favorites 6-43
setting 6-36
setting from favorites 6-41
viewing expressions 6-38
Who should read this book viii
Windows
Options 10-8
Project Properties 3-19
Workspace Options 10-8
Windows login ID
creating the home directory 1-8
Workspace
ALL 10-11
see also Appendix A
ALL group A-8
Asm_type setting A-7
Backup group A-11
Backup settings A-11
Board_file setting A-4
Button group A-6
Button settings A-6
changing settings 10-13
closing 10-5
Index

- closing objects 10-5
- Cmds group A-12
- CODE 10-11
  see also Appendix A
- CODE group A-6
- Command group A-2
- Command settings A-2
- connection details 10-12
- copying entries 10-15
- creating an empty file 10-7
- cutting entries 10-17
- DEBUGGER 10-11
  see also Appendix A
- DEBUGGER group A-2
- debugger settings 10-14
- defining Code windows 10-14
- deleting settings 10-17
- Disassembler group A-3
- Disassembler settings A-3
- Edit group A-10, A-12
- Edit settings A-12
- global options 10-2, 10-11
- in Tools menu 10-8
- in Workspace Options window 10-8
- initializing 10-2
- List of Entries pane 10-10
- not using 10-2
- open projects 10-6
- opening 10-3, 10-4
- pasting entries 10-15
- Pos_size group A-6
- Pos_size settings A-6
- project load list 10-6
- PROJECTS 10-11
  see also Appendix A
- resetting entries 10-17
- restoring settings 10-13
- same on start-up 10-4
- saving 10-3
- Search group A-10
- Search settings A-10
- settings 10-2, 10-11
  see also Appendix A
- settings conflict 10-5, 10-11
- settings file 10-11
- settings saved 10-2
- Settings Values pane 10-10
- Source_coloring group A-8
- Source_coloring settings A-8
- start-up options 10-3
- Tab_conv group A-11
- Tab_conv settings A-11
- Text group A-8
- Text settings A-9
- undoing changes 10-17
- using 10-2
- viewing settings 10-8
- window internals 10-11
- Workspace menu
  in the Code window 10-3
- Workspace Options window 10-9